Ray Tracing Under The Microcomputer

Abstract

Author Eugene Curran

The history of computer graphics is as old, almost, as the history of computing itself Until recently however, it had tended to be confined to the realm of the "aristocracy" of computer machines because of its requirement of their high power and speed as well as the additional cost of expensive graphics hardware In the last number of years however, a significant reduction in the price/performance ratio of both graphics and microprocessor technology has brought computer graphics within the grasp of the of the ordinary "working class" PC (Personal Computer) The nett result of this has been an increase in the number of users of computer graphics and its areas of application

One of these areas, that of the of generation of realistic three-dimensional images, is the subject matter of this work More specifically, this work is concerned with a particular method of generation of such images, known as *Raytracing*, which has produced some of the most realistic computer generated images to date Unfortunately, because of a large appetite for numeric calculation, raytracing has tended to be restricted to mainframe computers The motivation behind this research has been to implement a raytracing algorithm on a microcomputer and investigate its performance

Chapter one gives a general introduction to the area of computer graphics, while chapter two outlines a description of the raytracing algorithm, its advantages, limitations and optimizing techniques Chapter three then goes on to discuss the application of raytracing to the area of solid modelling and sets the context for the description of the research in chapter four, which discusses the design, implementation and performance of MicroTrace, a microcomputer based raytracer Finally, Chapter five discusses conclusions from the research and possible future enhancements to the work

Ray Tracing Under The Microcomputer

A THESIS SUBMITTED BY EUGENE CURRAN
FOR THE DEGREE OF Master of Science
AUGUST 1989

Ray Tracing Under The Microcomputer

A Thesis by EUGENE CURRAN B A I
Supervisor DR M SCOTT PHD

Submitted to
DUBLIN CITY UNIVERSITY
COMPUTER APPLICATIONS
for the degree of
Master of Science
August 1989

Declaration No portion of this work has been submitted in support of an application for another degree or qualification in the Dublin City University or any other University or Institute of Learning

Acknowledgement

To my parents, for the chance to pursue these studies, and to Dr Scott, for his advice and guidance along the way

Contents

List of figures

Ray Tracing Under The Microcomputer

Abstract

Author Eugene Curran

The history of computer graphics is as old, almost, as the history of computing itself Until recently however, it had tended to be confined to the realm of the "aristocracy" of computer machines because of its requirement of their high power and speed as well as the additional cost of expensive graphics hardware In the last number of years however, a significant reduction in the price/performance ratio of both graphics and microprocessor technology has brought computer graphics within the grasp of the of the ordinary "working class" PC (Personal Computer) The nett result of this has been an increase in the number of users of computer graphics and its areas of application

One of these areas, that of the of generation of realistic three-dimensional images, is the subject matter of this work More specifically, this work is concerned with a particular method of generation of such images, known as *Raytracing*, which has produced some of the most realistic computer generated images to date Unfortunately, because of a large appetite for numeric calculation, raytracing has tended to be restricted to mainframe computers The motivation behind this research has been to implement a raytracing algorithm on a microcomputer and investigate its performance

Chapter one gives a general introduction to the area of computer graphics, while chapter two outlines a description of the raytracing algorithm, its advantages, limitations and optimizing techniques Chapter three then goes on to discuss the application of raytracing to the area of solid modelling and sets the context for the description of the research in chapter four, which discusses the design, implementation and performance of MicroTrace, a microcomputer based raytracer Finally, Chapter five discusses conclusions from the research and possible future enhancements to the work

Chapter 1

An Introduction To Computer Graphics

1.1 Motivation

One of the principle advantages of Computer Graphics is its ability to present information in a *visual* form — a form which allows our well developed eye-brain pattern recognition mechanism to perceive and process the information more rapidly In this respect, the most frequent use of graphics today is probably to draw histograms, pie-charts, and two-dimensional or three-dimensional graphs of various mathematical and economic functions

However, Computer Graphics does play an essential role in many other widely varying fields, such as computer simulation, animation, exploration maps for drilling and mining, computer aided design and manufacture, art advertising and a profusion of others

The application of Computer Graphics to some of these areas, particularly flight simulation and animation, requires images capable of incorporating shadows, reflection/refraction of light, removal of hidden surfaces and shading in order to make them as true to life as possible *Raytracing* is the most successful method to date of incorporating all of these features into a graphics image and is the topic of subsequent chapters The concern of this chapter is to introduce, to the newcomer to computer graphics, the basic ideas and concepts involved in the generation of graphics pictures, which will enhance the understanding of subsequent chapters and facilitate a comparison raytracing with more traditional

graphics techniques

1.2 What Is Computer Graphics ?

Computer Graphics involves the generation of an image of an object from an appropriately defined description of the object It is this emphasis on the *synthesis* of pictures of real or imaginary objects that distinguishes Computer Graphics from the related field of image/picture processing The latter, which is important in areas such as satellite photograph enhancement and chromosome scans, is primarily concerned with the *analysis* of a picture and reconstruction of 2D or 3D objects from their pictures — the converse process of computer graphics

Since pictures generated using Computer Graphics will ultimately be seen on some sort of display device, before looking at the means by which such pictures are generated, some knowledge of the different display technologies will prove useful

1.3 Display Devices

The most common device for display of graphic output is the *Cathode Ray Tube* (CRT) The basic principle behind a CRT is that when a beam of electrons strikes a phosphor coated screen light is emitted The emitted light, however, decays exponentially with time so the process must be repeated many times per second (30 to 60) in order that the light appears unflickering to the viewer

The two principle categories of Cathode Ray Tube devices used in computer graphics are -

- Vector devices
- Raster devices

1.3.1 Vector Devices

As illustrated in *fig 1 1*, a vector device consists of a cathode ray tube, a display processor and a refresh buffer containing commands for plotting points, lines and characters These commands are interpreted by a display processor which converts the digital values into analog voltages which are used to electrostaticly displace the electron beam along the desired path, striking the phosphor and emitting light in the process

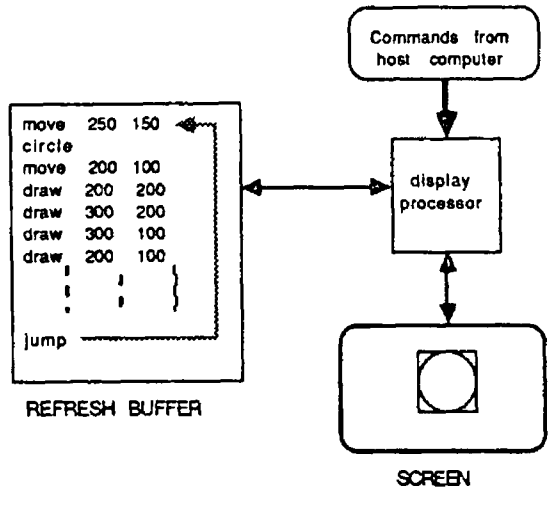


figure 1 1 A vector display device

Since the emitted light decays in something of the order of several hundred microseconds, the picture must be continually re-drawn using the commands in the refresh buffer (at least 30 times per second) in order that the picture does not appear to flicker to the viewer The time taken to re-draw the picture however is proportional to the number of lines in it so, where a picture has many lines that cannot be draw in less than $\frac{1}{30^{th}}$ of a second, flicker becomes unavoidable

Vector devices have the advantage of having very high resolution, typically 4096 screen dots horizontally by 4096 vertically, and a relatively small refresh buffer requirement (2K - 30K) Their principle drawbacks however are that they cannot display solid areas on screen and have only a very limited capability for displaying colour They are also expensive in comparison to raster devices and are not suitable for raytracing, as will become clear in chapter two

1.3.2 Raster Devices

The arrival, in the mid-seventies, of cheap raster graphics devices based on television technology and having a good capability for colour contributed greatly to the development of Computer Graphics and today, raster technology is the one most commonly found in graphics display devices

As *fig 1 2* illustrates, an image on a raster device consists of a rectangular matrix of points called pixels (short for picture element) Unlike the refresh buffer of a vector device which stores screen coordinates, a raster refresh buffer stores the *intensity* of each individual screen pixel The image is then drawn by sequentially scanning out each horizontal line of the buffer from left to right, top to bottom, to the screen The intensity for each pixel is determined by converting the value for the pixel stored in the buffer into an analog voltage that controls the intensity of the electron beam at that point on the screen

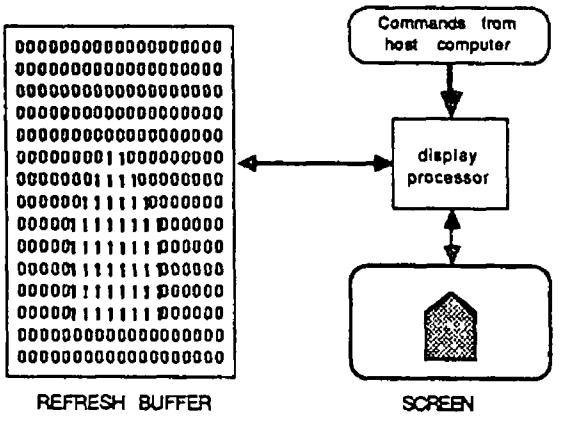


figure 1 2 A raster display device

In contrast to a vector device, where a line is stored as two screen coordinate values in the refresh buffer, a line is drawn on a raster device by calculating all screen pixels that the line will cross when drawn on the screen, and setting those pixel values accordingly in the refresh buffer There exists a number of efficient scan-line algorithms for performing this task not just for lines but for circles and other primitives as well (see [FOLE84])

In the simplest raster devices, the refresh buffer uses one binary bit per pixel to represent its intensity (1=ON, 0=OFF) Using more bits per pixel will permit a greater range of intensities in the screen image, at the expense of a

larger refresh buffer requirement

COLOUR In colour raster devices each pixel is actually composed of three phosphor dots, called a *triad* One of the dots emits red light when excited by an electron beam, the second green and the third blue These three colours, known as the primary colours, are used because almost any colour from the visible spectrum can be obtained from a suitable combination of them Three electron guns, arranged in the same triangular pattern as the triads, are used to synchronously excite the three phosphor colours so that when viewed from a distance, the triad will appear as a single dot whose colour is a mixture of the three dots For such devices three intensity values per pixel must be stored, corresponding to the intensity of each of the red, green and blue components of the pixel For example, storing two bits per primary (6 bits per pixel) allows 4 intensities per primary, giving a total of 64 colours

The principle advantages of a raster over a vector device are that it is less expensive, has a good capability for displaying colour and can display solid areas on screen On the other hand, raster devices do not yet have the same resolution as vector ones, 1284×1024 being considered high for raster, and have a much greater refresh buffer requirement, particularly for colour devices *e g* a colour raster display with 512×512 resolution and 64 colours per pixel requires a refresh buffer of 196608 bytes ($512 \times 512 \times 6$ *bits*) However, with the ever decreasing cost of memory, this drawback becomes less significant

1.4 Two Dimensional Graphics

Two dimensional objects exist in a completely flattened world They have length and width (and consequently area), but no thickness or volume, and are usually defined to the graphics system in terms lines, polygons, planar curves *etc* Yet, despite the limitation of two dimensions, this area of graphics is still very useful, both in its own right and as a stepping stone towards an understanding of the discussion of three dimensional graphics in *section 1 5*

1.4.1 Windows And Viewports

As will be seen from the following sections, several different coordinate systems are used in computer graphics The most basic one perhaps is the *screen* coordinate system This is a 2D integer coordinate system whose values in the X and

Y directions range from zero to the horizontal and vertical screen resolution size respectively, each coordinate pair directly corresponding to an actual pixel on the screen The position of the origin $(0,0)$ can vary from one device to another but is usually one of, the upper left corner, lower right corner, or centre, of the screen

It would be far too restrictive to use this coordinate system to directly define elements of a graphics picture as, not only would the definition have to be altered in order to be displayed on a device with different resolution, but the coordinate range would be inappropriate for many applications The solution to the problem is to use a coordinate system that is independent of screen coordinates, called a *world* (or virtual) coordinate system The Cartesian XY coordinate system is normally used as this virtual coordinate system

The approach is to use this virtual coordinate system to define objects and then "map" the the virtual coordinates onto screen coordinates Thus, the object definition remains independent of the display device and only the mapping changes from one device to another Mapping the entire virtual space onto the screen however would mean that only very large objects would be visible A rectangular area called a *window*, defined by four virtual coordinate values $(X_{min}, X_{max}, Y_{min}, Y_{max})$, is therefore used to specify a section of the virtual coordinate space to map onto the screen

Similarly, instead of always mapping the window onto the entire screen, a greater degree of flexibility in displaying the picture is possible if the window can be mapped onto a specified sub-region of the screen This would allow, for example, several windows to be displayed simultaneously on different areas of the screen A *viewport* is therefore used to define a rectangular area of the screen onto which the window is to be mapped As illustrated in *fig 1 3*, this window-viewport combination allows any section of the virtual coordinate space to be displayed on any area of the screen

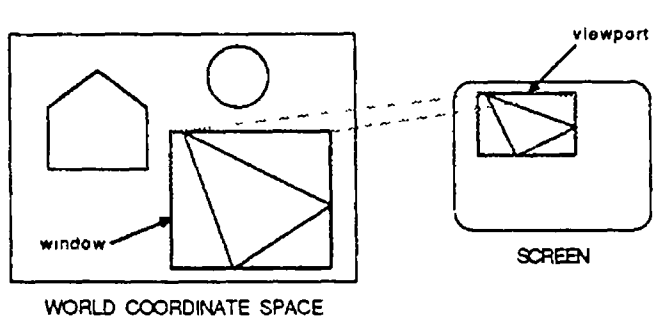


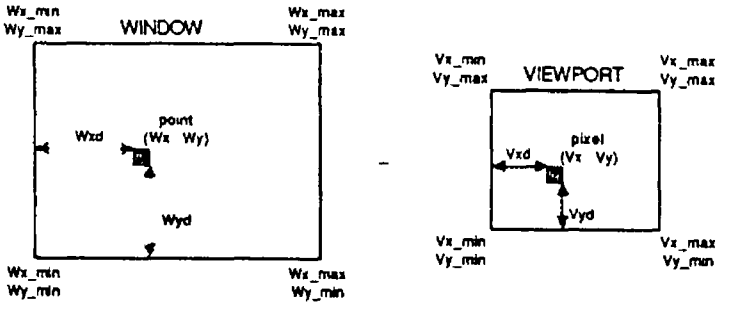
figure 1 3 Window to viewport mapping

MAPPING From *fig 1 4*, it can be seen that the equations which map a point (W_x, W_y) inside the window onto a point (V_x, V_y) inside the viewport are given by -

$$V_x = (W_x - W_{xmin})\frac{V_{xmax} - V_{xmin}}{W_{xmax} - W_{xmin}} + V_{xmin}$$

$$V_y = (W_y - W_{ymin})\frac{V_{ymax} - V_{ymin}}{W_{ymax} - W_{ymin}} + V_{ymin} \qquad (1\ 1)$$

where the window is defined by $(W_{xmin}, W_{xmax}, W_{ymin}, W_{ymax})$ and the viewport by $(V_{xmin}, V_{xmax}, V_{ymin}, V_{ymax})$



preservation of horizontal ratios => $\frac{Wxd}{Wx\_max \quad Wx\_min} = \frac{Vxd}{Vx\_max \quad Vx\_min}$

$$\Rightarrow \frac{Wx \quad Wx\_min}{Wx\_max \quad Wx\_min} = \frac{Vx \quad Vx\_min}{Vx\_max \quad Vx\_min}$$

$$\Rightarrow Vx = (Wx \quad Wx\_min)\ \frac{Vx\_max \quad Vx\_min}{Wx\_max \quad Wx\_min} + Vx\_min$$

Similarly

$$Vy = (Wy \quad Wy\_min)\ \frac{Vy\_max \quad Vy\_min}{Wy\_max \quad Wy\_min} + Vy\_min$$

figure 1 4 Window to viewport mapping equation

This defines the mapping function for the window to the viewport In ap-

plying the function however, it must be ensured that points outside the window are not mapped, as such points would be mapped onto non-existent screen coordinates, resulting in a "wrap-around" effect whereby points mapped beyond the right of the screen are "wrapped around" and appear on the left of it A technique known as window clipping, which clips off those parts of the object outside the window from the mapping function, is therefore applied As illustrated in *fig 1 5*, there are three possible cases when clipping a line It can he -

- Entirely outside the window (line A)
- Entirely inside the window (line B)
- Partially inside the window (line C)

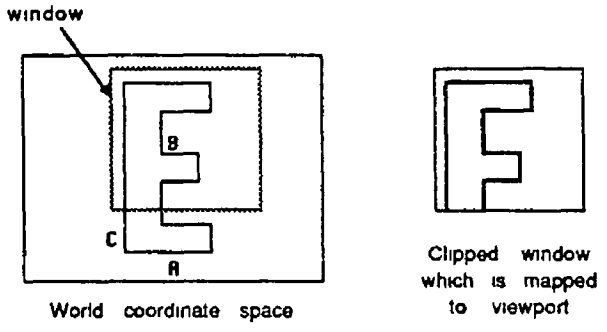


figure 1 5 Window clipping

Only the third case poses any difficulties since the window clipping operation must determine the intersection of the line with the edge of the window For a detailed discussion on window clipping algorithms, such as the Cohen-Sunderland algorithm, see Foley & Van Dam [FOLE84]

1.4.2 2D Matrix Transformations

As mentioned in the previous section, there are good reasons for defining objects using a virtual coordinate system instead of directly using screen coordinates Having defined the object in such a coordinate space, many graphics applications

require that the object can be moved around the virtual space or have its orientation changed, in much the same way as an object in the real world Fortunately, there are mathematical transformations which provide a means of of translating (repositioning) and rotating (reorientating) objects In addition to these two operations, there exists a scaling transformation that can be applied to make an object larger or smaller, something which cannot be done with real objects From the following discussion of the mathematics of these three transformations, it can be seen that the use of homogeneous coordinates allows each transformation to be represented as a 3×3 matrix — a result which proves very useful and which is discussed at the end of this section

Homogeneous coordinates are coordinates that were developed in geometry by Maxwell [MAXW46], and later applied to computer graphics by Blinn & Newell [NEWE78] A 2D Cartesian point $P(x,y)$ is represented in homogeneous coordinates as $P(xW,yW,W)$ where W is some non-zero scale factor So, given a homogeneous coordinate point $P(X,Y,W)$, its $2D$ Cartesian representation $P(x,y)$ is given by $x = \frac{X}{W}$, $y = \frac{Y}{W}$

$$P(X,Y,W) \quad = \quad P(\frac{X}{W},\frac{Y}{W}) \quad = \quad P(x,y)$$

Similarly, for three dimensions

$$P(X,Y,Z,W) \quad = \quad P(\frac{X}{W},\frac{Y}{W},\frac{Z}{W}) \quad = \quad P(x,y,z) \qquad (1\,2)$$

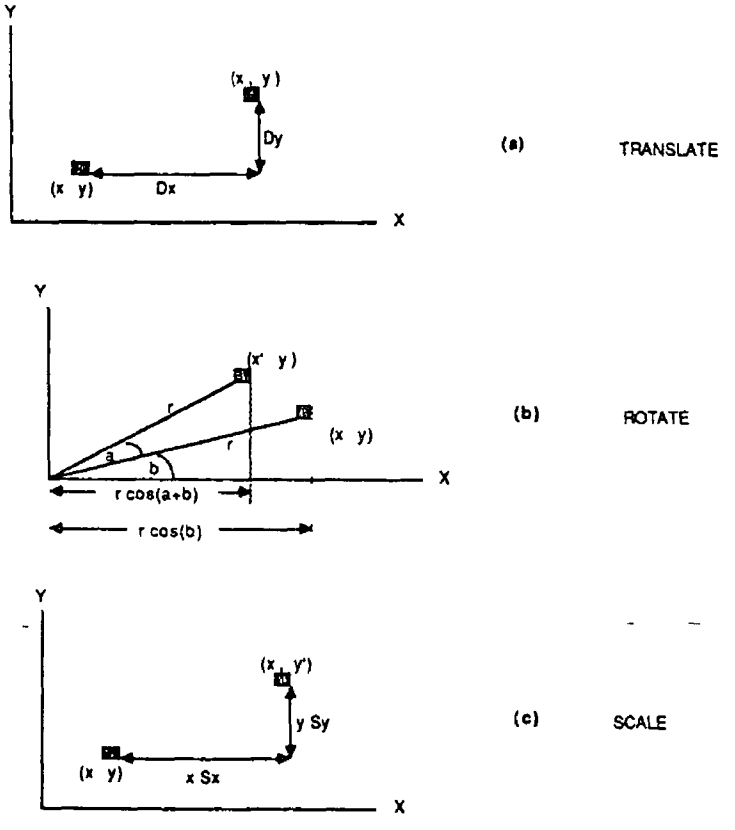
Since $W = 1$ throughout this section there is no need to perform the division It is only in the perspective transformation matrix in *section 1 5 2* that a value other than 1 is obtained and a division by W has to be performed

TRANSLATION A point $P(x,y)$ is translated to new point $P'(x'y')$ by the addition of a displacement D_x units parallel to the x-axis and D_y units parallel to the y-axis, *fig 1 6a* This can be expressed in vector form as

$$[x',y'] = [x,y] + [D_x,D_y] \qquad (1\,3)$$

Rewriting this in homogeneous coordinates means the translation can be represented as a 3×3 matrix (the reason for using this form will become clear later)

$$[x', y', 1] = [x, y, 1] \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ D_x & D_y & 1 \end{bmatrix} \tag{1 4}$$



Translation Rotation and Scaling transformations

figure 1 6

ROTATION *Fig 1 6b* illustrates the anti-clockwise rotation, by alpha degrees, of a point $P(x, y)$ about the origin to a new point $P'(x', y')$ From the diagram it can be seen that -

$$x = r \cos b$$

$$y = r \sin b \tag{1 5}$$

and

$$x' = r \cos(a + b) = r \cos b \cos a - r \sin b \sin a$$

$$y' = r\sin(a+b) = r\cos b\sin a - r\sin b\cos a \tag{1 6}$$

Substituting (1 5) into (1 6) gives

$$\begin{aligned} x' &= x\cos a - y\sin a \\ y' &= x\sin a + y\cos a \end{aligned} \tag{1 7}$$

Using homogeneous coordinates, the rotation can be represented as a 3×3 matrix, and (1 7) can be written as

$$[x', y', 1] = [x, y, 1]\begin{bmatrix} \cos a & \sin a & 0 \\ -\sin a & \cos a & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1 8}$$

For this derivation, positive angles were measured in an anti-clockwise direction By substituting the identities $\cos(-a) = cos(a)$ and $\sin(-a) = -sin(a)$ into (1 7) and (1 8), positive angles can be measured in a clockwise direction

SCALING A point $P(x, y)$ can be scaled by S_x along the x-axis and S_y along the y-axis, with respect to the origin, to a new point $P'(x', y')$, by the following multiplication, *fig 1 6c*

$$\begin{aligned} x' &= xS_x \\ y' &= yS_y \end{aligned} \tag{1 9}$$

The scaling can be represented as a 3×3 matrix by writing the coordinates in homogeneous form

$$[x', y', 1] = [x, y, 1]\begin{bmatrix} Sx & 0 & 0 \\ 0 & Sy & 0 \\ 0 & 0 & 1 \end{bmatrix} \tag{1 10}$$

COMPOUND TRANSFORMATIONS It will often be the case that more than one of these transformations will want to be performed on an object For example, rotating an object about a point other than the origin, say $R(x, y)$, involves translating R to the origin, rotating about the origin, then translating back again Each of these three transformations must be performed on each point

on the object If however, each transformation is represented as a 3×3 matrix, the three operations can be compounded into one by multiplying the three matrices together, resulting in a new 3×3 matrix that represents the compound operation Multiplying each point on the object by this new matrix applies the compound transformation in one operation, with a resultant computational saving

1.5 The Third Dimension

The addition of a Z-axis to the XY virtual space of 2D graphics allows objects to take on depth and volume, but brings with it the complication of trying to display a *three* dimensional entity on a *two* dimensional screen, in addition to the that of trying to determine if one object in a scene obscures all or part of another — two complications which are discussed in *sections 1 5 2 and 1 7 1* respectively

However, it is comforting to know that the translation, rotation, and scaling transformations of the previous section can still be represented in matrix form when extended to three dimensions Before discussing each of these transformations, it is first worth noting from *fig 1 7* that there are in fact two possible directions in which the positive Z-axis can be faced, giving rise to two different coordinate systems The right handed system has the Z-axis pointing in the direction of the vector cross product of the X-axis with the Y-axis *i e* out of the page, and is the system used throughout this text, while the left handed one has it pointing in the opposite direction

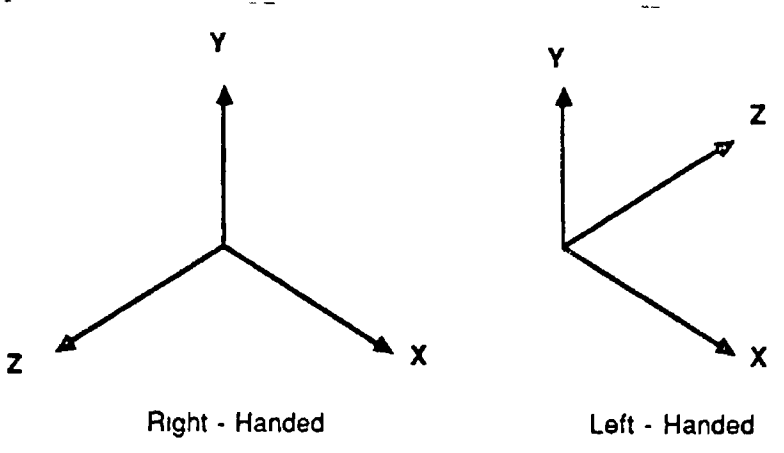


figure 1 7 Right and Left Handed Coordinate Systems

1.5.1 3D Matrix Transformations

Section 1.4.2 illustrated how each of the two dimensional translation, rotation and scaling transformations could be represented as a 3×3 matrix. The following section outlines their extension to three dimensions, where it can be seen that each can be defined by a 4×4 matrix.

TRANSLATION Using homogeneous coordinates, the translation of a point $P(x, y, z)$ to another point $P'(x', y', z')$ by displacements of D_x, D_y, D_z parallel to the X, Y, and Z axes can be represented by the 4×4 matrix in the following equation

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ D_x & D_y & D_z & 1 \end{bmatrix} \tag{1.11}$$

ROTATION In three dimensional graphics, three different rotations (one about each of the three principle axes) can be performed, each with their own form of 4×4 matrix. The positive direction of rotation about an axis is defined as that which is anti-clockwise when looking down the positive part of the axis toward the origin. The 4×4 matrix representation for the rotation of a point $P(x, y, z)$ by an angle α to a new point $P'(x', y', z')$ is given for each type of rotation in *equations 1.12 to 1.14* below.

ROTX

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & \cos\alpha & \sin\alpha & 0 \\ 0 & -\sin\alpha & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.12}$$

ROTY

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} \cos\alpha & 0 & -\sin\alpha & 0 \\ 0 & 1 & 0 & 0 \\ \sin\alpha & 0 & \cos\alpha & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.13}$$

ROTZ

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} \cos\alpha & \sin\alpha & 0 & 0 \\ -\sin\alpha & \cos\alpha & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.14}$$

SCALING *Equation 1.15* below gives the 4×4 matrix for scaling a point $P(x, y, z)$ about the origin by factors of S_x, S_y, S_z, parallel to the X, Y, and Z axes, to a new point $P'(x', y', z')$. Scaling a point about a point other than the origin, say $R(a, b, c)$ is done by translating the point by $(-a, -b, -c)$, scaling it, then translating the scaled point by (a, b, c).

$$[x', y', z', 1] = [x, y, z, 1] \begin{bmatrix} Sx & 0 & 0 & 0 \\ 0 & Sy & 0 & 0 \\ 0 & 0 & Sz & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1.15}$$

COMPOUND TRANSFORMATIONS As with 2D transformations, multiple transformations can be performed on a point by the multiplication of a single 4×4 matrix representing the compound transformations — the latter being formed by multiplying together the transformation matrix for each of the transformations comprising the compound one. It should be noted however, that such compound transformations are not generally commutative *i.e.* the order in which rotation, scaling and translation is performed is significant.

Such compound transformations allow objects to be defined in their own local coordinate system and then transformed into the world (or some other intermediate) coordinate system. For example, the car wheels in *fig 1.8* are each defined in their own local coordinate systems which are then transformed into appropriate locations of the car coordinate system, which in turn is transformed into the world coordinate system.

1.5.2 3D Geometric Projections

Viewing an object in 2D simply involves specifying a window on the virtual 2D view-plane, a viewport on the screen and directly mapping one onto the other (*section 1.4.4*). The 3D viewing process however is inherently more complex by

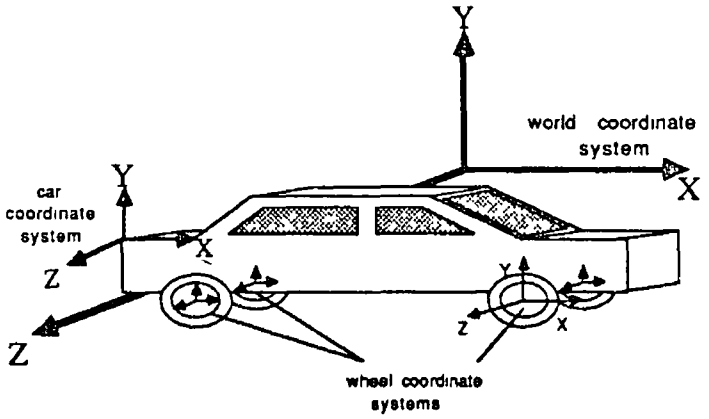


figure 1 8 Multiple coordinate systems

virtue of the fact that it involves the display of three dimensional objects on a two dimensional display device

This difficulty is overcome by the use of *planar geometric projections* This firstly involves projecting the 3D objects onto a 2D projection plane, then mapping this plane onto the screen in the same way as the virtual plane of 2D graphics (*figs 1 9 and 1 11* give examples of two different types of projection) Just as objects in 2D graphics are clipped against a window before being mapped onto the viewport, objects in 3D are clipped against a *view volume* before being projected onto the projection plane The projection onto the projection plane of the view volume itself then serves as a window to map onto the viewport While in the most general case, the projection plane can be any arbitrary plane, the XY plane is used throughout the following discussion since this leads to a simplification of the mathematics of the two main categories of projection outlined The more general case is discussed in [FOLE84]

Many different types of projection can be used in projecting an object onto a 2D projection plane The type of projection used will determine what the object finally looks like when it is seen on the screen The types of projection most commonly used in 3D graphics can be divided into the following two general catagories -

[ ] Parallel projections

[ ] Perspective projections

PARALLEL PROJECTIONS In this type of projection, *fig 1 9*, the lines

of projection are parallel to each other *i e* the centre of projection is at *infinite* distance from the projection plane The projection is defined by a direction of projection, and is classified as orthographic or oblique depending on whether or not this direction is orthogonal to the projection plane Common orthographic projections are the front (elevation), plan (top), and side (elevation) projections, which project onto the XY, XZ, and YZ planes respectively and mathematiclly, are the simplest projections to perform *e g* orthographic projection of a point $P(x, y, z)$ onto the XY plane simply involves "chopping off" the Z coordinate, giving $P'(x, y)$ as the projection plane coordinate of the point

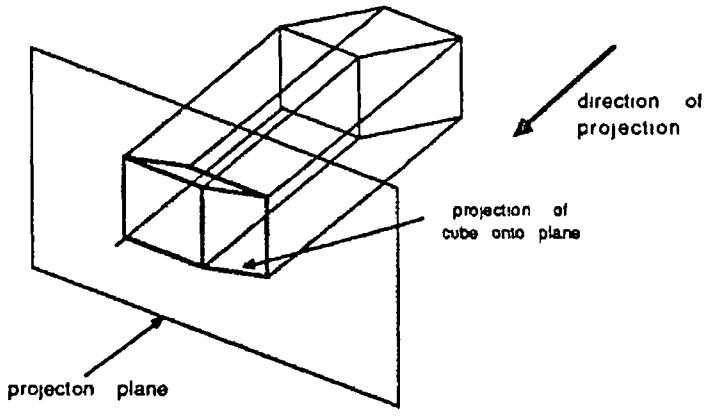


figure 1 9 PARALLEL PROJECTION

Two common oblique projections are the cavalier and cabinet projections, where the direction of projection makes an angle of 45 degrees and arctan $\frac{1}{2}$ respectively with the projection plane The mathematics for an oblique projection onto the XY plane specified by a direction vector $D(a, b, c)$ is given below -

The parametric equation of a line with a direction of $D(p, q, r)$ and containing a point $P(X_0, Y_0, Z_0)$ is given by the following equation (where t takes on values from minus infinity to plus infinity) [ANTO81] -

$$\begin{aligned} X &= X_0 + tp \\ Y &= Y_0 + tq \\ Z &= Z_0 + tr \end{aligned} \qquad (1\ 16)$$

Hence, from *fig 1 10*, the equation of the projector through $P(X_p, Y_p, Z_p)$ is given

by -

$$\begin{aligned} X &= X_p + tA \\ Y &= Y_p + tB \\ Z &= Z_p + tC \end{aligned} \tag{1 17}$$

which intersects the XY plane ($Z = 0$ plane) at $t = -\frac{Z_p}{c}$

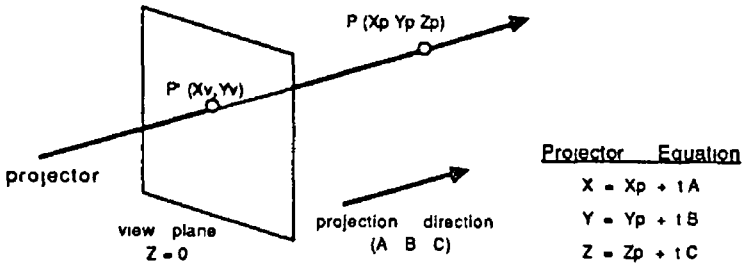


figure 1 10 Projector equation for a parallel projection

So, $P'(X_v, Y_v)$, which is the parallel projection of $P(X_p, Y_p, Zp)$ onto the XY plane in the direction of $D(a, b, c)$ is given by -

$$\begin{aligned} X_v &= X_p - Z_p\frac{a}{c} \\ Y_v &= Y_p - Z_p\frac{b}{c} \end{aligned} \tag{1 18}$$

Using homogeneous coordinates, the projection can be expressed as a 4×4 matrix -

$$[X_v, Y_v, Z_v, 1] = [X_p, Y_p, Z_p, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ -\frac{a}{c} & -\frac{b}{c} & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1 19}$$

NOTE $Zv = 0$

This format proves very useful since it means that the projection can be incorporated into the transformation matrix for an object by multiplying the two 4×4 matrices

PERSPECTIVE PROJECTIONS Perspective projections, *fig* 1 11, have a centre of projection that is a *finite* distance from the projection plane Unlike parallel projections however, perspective projections produce a perspective foreshortening effect (objects further from the centre of projection appear smaller) and hence produce a greater degree of realism, since this effect is also experienced by the human visual system

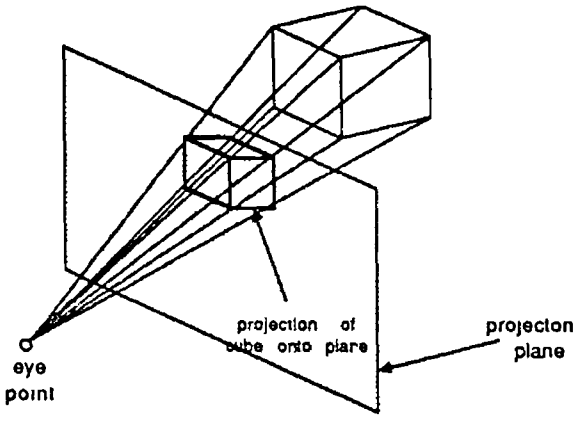


figure 1 11 PERSPECTIVE PROJECTION

The mathematics for a perspective projection onto the XY plane specified by a centre of projection on the positive Z-axis, a distance d from the origin, is outlined below -

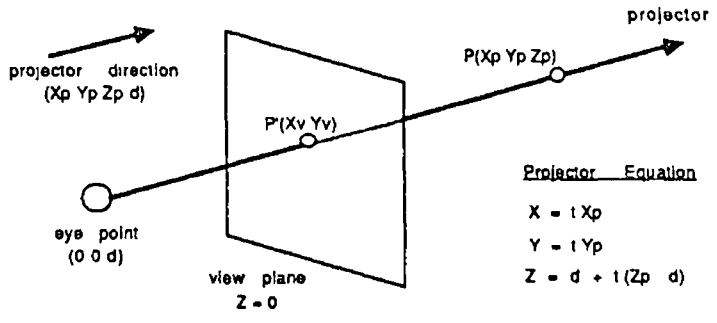
From *fig 1 12* the equation of the projector is that of a line containing $P(0,0,d)$ and having a direction of -

$$(X_p, Y_p, Z_p) - (0,0,d) = (X_p, Y_p, Z_p - d) \tag{1 20}$$

So, from (1 16), the equation of the projector is given by

$$\begin{aligned} X &= tX_p \\ Y &= tY_p \\ Z &= d + t(Z_p - d) \end{aligned} \tag{1 21}$$

and the projector intersects the XY plane at $Z = 0$ *i.e* at $t = \frac{1}{1-\frac{Z_p}{d}}$



Projector equation for a perspective projection

figure 1 12

Therefore, $P'(X_v, Yx)$, the perspective projection of $P(X_p, Y_p, Z_p)$ onto the XY plane, is given by -

$$\begin{aligned} X_v &= \frac{X_p}{1 - \frac{Z_p}{d}} \\ Y_v &= \frac{Y_p}{1 - \frac{Z_p}{d}} \end{aligned} \tag{1 22}$$

As with the parallel projection, using homogeneous coordinates allows this perspective projection to be specified as a 4×4 matrix, which means that the projection of an object can be incorporated into its transformation matrix

$$[X, Y, Z, W] = [X_p, Y_p, Z_p, 1] \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & -\frac{1}{d} \\ 0 & 0 & 0 & 1 \end{bmatrix} \tag{1 23}$$

where $[X, Y, Z, W] = [X_p, Y_p, 0, 1 - \frac{Z_p}{d}]$

However, to return $[X, Y, Z, W]$ to the form $[X_v, Y_v, Z_v, 1]$ a division by W is required (see *equation 1 2*)

$$\left[\frac{X}{W}, \frac{Y}{W}, \frac{Z}{W}, 1\right] = [X_v, Y_v, Z_v, 1] = \left[\frac{X_p}{1 - \frac{Z_p}{d}}, \frac{Y_p}{1 - \frac{Z_p}{d}}, 0, 1\right] \tag{1 24}$$

1.6 Object Representation

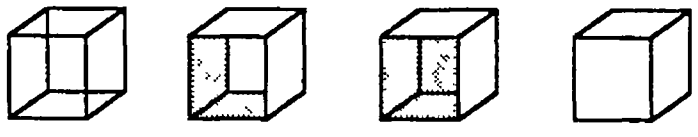
There are two types of situation in which the need to represent 3D shapes arises The first case is when an existing object such as a car, house or mountain is to be represented as a 3D graphic object, and the second, which frequently occurs in computer aided design, is where a designer interactively builds up an imaginary 3D object on computer from some preliminary sketches of what the object should finally look like In the first case, the computer representation of the object should try to match as closely as possible the exact shape of the real object, and in the second, should allow the designer to easily manipulate the object so that it can be molded into the desired form

While the previous sections have looked, in some detail, at the means by which objects defined in a virtual coordinate system can be scaled, rotated, and translated, and then mapped onto screen coordinates to form a final screen image, little has been said about the means by which such objects can be defined to the graphics system The remainder of this section looks at two contrasting object representation schemes commonly used in 3D computer graphics, *polygon mesh* and *Constructive Solid Geometry* (CSG) representations The first represents solid objects as closed surfaces which are defined as a collection of polygons while the second, the one used in this research, represents them directly as solids formed from by the addition and subtraction of basic solids (called primitive solids) such as spheres, cubes, cones and cylinders

1.6.1 Polygon Mesh Representation

Representing an object as a collection of lines means that only a line display of the object can be generated In addition, weight or volume calculations cannot be performed on the object The reason for this is that lines alone do not define surfaces (see *fig 1 13*), and it is surfaces that are required to perform hidden surface removal, volume calculations, *etc*

A polygon however can define a bounded planar surface, and a group of such polygons, called a polygon mesh, can be used to define the surfaces of some object Polygons provide a good means of representing objects that are composed of many flat surfaces, such as buildings, tables, desks *etc* They can also be used to represent objects with curved surfaces by approximating the curved part as a collection of small polygons, but this gives only an approximate representation The error of approximation can however, be made arbitrarily small by using larger



The same set of 12 lines in (a) can represent any of the three objects in (b) (c) or (d) A higher level primitive a surface is therefore required to unambiguously represent 3D objects

figure 1 13

numbers of smaller polygons to represent the object, but this will increase both the storage space requirements of the object representation, and the execution time of any algorithms that process it

1.6.2 Constructive Solid Geometry (CSG)

In the polygon mesh representation scheme a solid object is modelled, not as a solid, but as a closed surface The CSG representation scheme however models solid objects as compositions of primitive solids that are combined using boolean set operators The advantages of such a representation are -

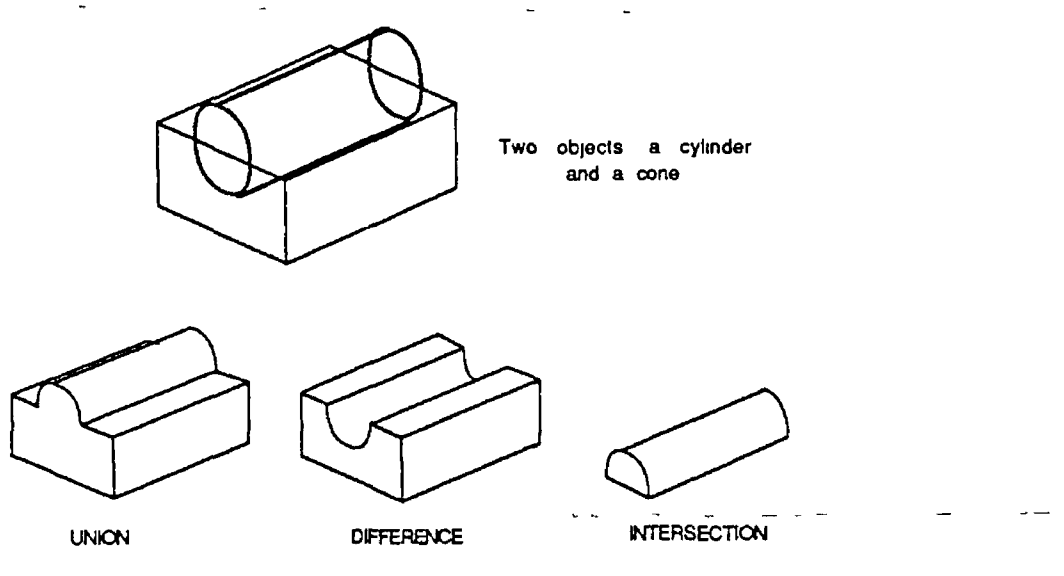
[ ] The model represents a true solid with volume

[ ] Solids are bounded by both curves and planar surfaces

[ ] Mechanical parts can be particularly well represented

The following three boolean set operators, illustrated in *fig 1 14*, are used to combine primitive (and intermediate) solids -

- Union
- Intersection
- Difference



Union difference and intersection operations on two solids

figure 1 14

Union The space occupied by a solid defined as the union of two other solids, A and B, is the space occupied by solid A plus the space occupied by solid B

Intersection The space occupied by a solid defined as the intersection of two other solids, A and B, is the space occupied by solid A that is also occupied by solid B

Difference The space occupied by a solid defined as the difference of two other solids, A and B, is the space occupied by solid A, less any part of that space that is also occupied by solid B

The primitive solids normally used are blocks/cubes, spheres, cones, and cylinders but others, such as superquadric surfaces [EDWA82] can be used Solids

formed as a result of the combination of two such primitives can themselves be combined, and in this way, representation of more complex solids can be built up in the form of a binary tree where the leaf/end nodes are primitive solids, the root node represents the entire solid, and the intermediate-nodes represent intermediate solids (see *section 3 2 1*)

1.7 Adding Realism

As mentioned at the start of this chapter, there are some areas where the application of computer graphics does not not simply require a screen image of an object, but an image that looks as real to the viewer as possible Adding this realism to the picture involves such techniques as removing hidden surfaces, shading objects, incorporating shadows, and adding texture to surfaces

1.7.1 Hidden Surface Removal

Hidden surface removal involves determining which objects in a picture are visible to the viewer and which are obscured by other objects, given a particular viewing point, projection type, projection plane *etc* Although the idea sounds quite simple, the reality is that its implementation requires such large effort of computation that many carefully considered algorithms have been developed Sutherland, Sproull and Schumacker [SUTH74] survey ten such algorithms and provide a good introduction to the topic The details of any particular algorithm will depend of course on the object representation scheme in use That is to say, an algorithm for removing hidden surfaces from a polygon mesh representation will be quite different from one that assumes say, a parametric bicubic patch object representation Some of the more commonly used algorithms are outlined below

DEPTH SORT The approach of this algorithm, which was developed by Newell, Newell and Sancha, is straightforward and simple The general idea is to draw all polygons in the scene, but to *sort* them beforehand so that polygons furthest from the viewer are drawn first In this way, if a polygon is obscured from the viewer by another polygon, the obscured polygon (being further from the viewer) will have been drawn first and will be overwritten by the obscuring one The three general steps of the algorithm are outlined below For a more detailed discussion see [NEWE72]

- Sort all polygons in the scene according to the largest z-coordinate of each
- Resolve any ambiguities that may arise from any overlapping polygons
- Scan-convert each polygon into the refresh buffer in descending order of largest z-coordinate

Z-BUFFER The z-Buffer algorithm, see [FOLE84], adopts a similar approach to the Depth Sort one, except that polygons can be scan converted into the refresh buffer in any order through the use of an additional buffer, called a z-buffer, which stores for each pixel, the Z value of the point on the polygon that currently covers that pixel Only if the Z value of the point on a subsequent polygon which also covers that pixel is less than the value stored in the z-buffer, is the pixel updated and the z-buffer value for that pixel changed to the new Z value

Other algorithms, developed by Bouknight [BOUK70] and Watkins [WATK70], also deal with removing hidden surfaces from objects defined by polygons While these algorithms, like the two above, can be applied to objects defined by curved surfaces by first approximating the surfaces with many small polygons, algorithms for dealing directly with curved surfaces have also been developed These include algorithms developed by Weiss [WEIS66], Mahl [MAHL72] and Levin [LEVI76] for dealing with objects defined by quadric surfaces, and algorithms by Catmull [CATM80] and Blinn [BLIN80] for parametricly defined surfaces

1.7.2 Shading

Having removed hidden surfaces through the use of one of the above algorithms, the visible surfaces (particularly curved surfaces) must be correctly shaded in order to give any sort of real effect For example, a sphere drawn without shading would appear as a flat circle on the screen In shading an object, the shading calculation should take into account such parameters as the position and orientation of both the light source(s) and the surface to be shaded, as well as surface characteristics (flat, smooth *etc*) and, in the case where specular reflection is

taken into consideration, the position of the viewer The light source can be either a point source, such as an incandescent bulb, or a distributed source, such as a bank of fluorescent lights Point sources however, are normally used since they result in the calculation of only a single vector from a point on a surface to the light source, whereas several are required to approximate a distributed source The exact details of the calculation will of course depend on the complexity of the lighting model, which can incorporate any or all of the following basic light components -

- Ambient light
- Diffusely reflected light
- Specularly reflected light

AMBIENT LIGHT Ambient light is a light of uniform brightness found in most real environments as a result of the multiple reflections of light from the many surfaces normally found in such environments, and is the simplest of the three components to model The amount of ambient light, I_A, reaching a viewer from a surface is given by -

$$I_A = I_a K_a \tag{1 25}$$

where I_a is the intensity of the ambient light, and K_a is the fraction of ambient light reflected by the surface

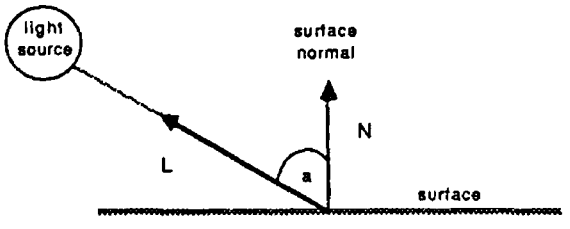
DIFFUSE REFLECTION This is the type of reflection exhibited by dull matte surfaces Such surfaces scatter light equally in all directions and consequently appear to have the same brightness from all viewing angles The intensity of diffusely reflected light, I_D, from a point on such a surface can be determined from Lambert's cosine law and is dependent on the cosine of the angle between the normal to the surface, $\vec{N}$, and the vector in the direction of the hght source $\vec{L}$, *fig 1 15* -

$$I_D = I_p K_d \cos a \tag{1 26}$$

where I_p is the intensity of the light source and K_d is a number between 0 and 1 called the *diffuse reflection coefficient* for the surface, which varies from one

surface to another If $\vec{N}$ and $\vec{L}$ are normalized, (1 16) can be rewritten using the vector dot product -

$$I_D = I_p K_d(\vec{L}\,\vec{N}) \tag{1 27}$$



Diffuse reflection intensity is proportional to angle a

figure 1 15

SPECULAR REFLECTION Look at any shiny surface and you will see a highlighted area *i e* an area that is significantly brighter than the rest of the object and emitting light that is nearer to the colour of the light source (usually white) than the colour of the object itself The highlight is a result of *specular reflection* and is due to the fact that shiny surfaces reflect light unequally in different directions The intensity of specularly reflected light emitted from such a surface rapidly falls off as angle b in *fig 1 16* increases

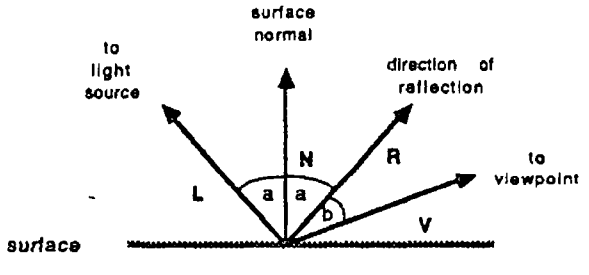


figure 1 16 Specular reflection

Phong Bui-Tuong approximated this rapid fall off as $\cos^n b$ [BUIT75], where n is typically between 1 and 200, depending on the smoothness of the surface (the higher the value, the shiner the surface) The approximation, which is quite accurate, is based on an empirical observation rather that any theoretical derivation In contrast, the Torrence-Sparrow shading model [TORR66], developed by

illumination engineers, is a theoretically based one, adapted to computer graphics by Blinn and compared with the Phong model in [BLIN77] The amount of light that is specularly reflected from a surface is also a function of the angle of incidence (angle a in *fig 1 16*) If this function is represented as $F(a)$, then, using the Phong model, the intensity of specularly reflected light, I_S, reaching a viewer from a surface is approximated by -

$$I_S = I_p F(a) \cos^n b \tag{1 28}$$

where I_p is the intensity of incident light To reduce computation of I_S, $F(a)$ is often set to a constant K_s, which is selected by trial and error to give the best results If $\vec{R}$ and $\vec{V}$ in *fig 1 16* are normalized, equation (1 28) can be written as -

$$I_S = I_p K_s (\vec{R}\,\vec{V})^n \tag{1 29}$$

Combining ambient, diffuse and specular components, the intensity of light reaching a viewer from a surface I, is the sum of the three components which, from (1 25), (1 27) and (1 29), can be written as -

$$I = I_a K_a + I_p \left[K_d (\vec{L}\,\vec{N}) + K_s (\vec{R}\,\vec{V})^n \right] \tag{1 30}$$

Even with the incorporation of all three light components (ambient, diffuse and specular), the above lighting model has certain limitations For example, the model does not take account of global illumination information, *i e* in calculating the light reflected from a point on a surface, it does not take account of light reflected from or refracted through other objects in the scene that may be incident on the surface Consequently, the reflection of one object in another object or the visibility of one object through a transparent object, will not be emulated in the final screen image Both Whitted [WHIT80] and Greenberg [GREE79] however, have implemented models that resolve this shortcoming Whitted's approach (which is a raytracing one) is computationally more expensive than Kay's (but is more general) and is based on an earlier raytracing algorithm by Appel [APPE68] Whitted's model is outlined in *section 2 3 2* of the next chapter, which deals with raytracing

As the quest for greater visual realism continues, even more complex lighting models are being developed, such as those by Cook [COOK81] [COOK88], Nishita and Nakamae [NISH86] for shading objects illuminated by natural sunlight, Max [MAX86] for dealing with atmospheric illumination, and Cohen and Greenberg's radiosity method [COHE85] of catering for diffuse reflection in complex environments

1.7.3 Shadows and Texture

While previous sections have looked at the problems of removing hidden surfaces from a scene, and the correct shading of objects, no mention has been made of the inclusion of shadows in a scene With the exception of the case where the view-point and the light source are in the same same location, the viewer of a scene will observe shadows cast by the objects Since the surfaces that lie in shadow are the ones that are visible from the viewpoint but not from the light source, some rendering systems calculate shadows by invoking the hidden surface algorithm twice, once for the viewpoint and a second time for the light source

Crow examines several ways of generating shadows for polygonal objects, [CROW77A], while Lance [LANC78] addresses the generation of curved shadows on curved objects In a variation of one of Crows algorithms, Greenberg, Atherton & Weiler [GREE78] incorporate shadows for polygonal objects by associating with each polygon that is either completely or partly visible from the light source, a secondary coplanar polygon that marks the area of the first one that is visible from the light source These secondary polygons are then used to indicate to the shading algorithm which parts of the main polygons lie in shadow (namely the area of the polygon minus that covered by the secondary polygon)

TEXTURE The shading algorithm described in *section 1 7 2*, when applied to either planar or curved surfaces, produces very smooth and uniform surfaces This is because there are actually two types of surface detail, colour and *texture*, and it is the latter one which gives a surface the roughened look characteristic of most of the surfaces of the real world Since it would be impractical to use polygons to attempt to model very fine levels of texture, Catmull [CATM74] pioneered a technique of mapping a digitized photograph of the texture of a real surface onto a computer generated surface, a technique which was later refined by Blinn and Newell [BLIN76] The technique involves mapping a pattern array, which represents the digitized texture photograph, onto a planar or curved surface, by a means similar to that used for pattern filling polygons (see [FOLE84])

A more recent technique applied by Carpenter [CARP82] and Mandelbrot [MAND82], uses a class of irregular shapes, called fractals, which are probabilistically defined and can accurately model natural shapes such as coastlines, mountains, snowflakes, tree branches *etc*

Chapter 2

An Introduction To Ray Tracing

Ray tracing is a very powerful yet simple approach to image synthesis which has generated some of the most realistic computer images to date It is capable of incorporating multiple reflections and refractions from multiple objects in a scene, can deal with multiple light sources, and can model effects such as penumbras, motion blur and other fuzzy phenomena that would prove difficult, if not impossible, with other image generation techniques

The technique was first suggested by Appel [APPE67] and was later used by Goldstein and Nagel [NAGE71] as a solution to the hidden surface problem But it wasn't until the late 1970's that it was implemented by Kay & Greenberg [GREE79] and by Whitted [WHIT80] to render complete images

2.1 The Ray Tracing Approach

From the discussion of computer graphics in the last chapter, it can be seen that the conventional approach to three dimensional graphics is to define a viewing point, a projection plane and a view volume, then project objects inside the view volume onto the projection plane in accordance with a perspective/parallel projection and map the projected coordinates onto screen coordinates The final image produced by such a process is an unrealistic wireframe view of the object/scene However, as outlined in *section 1 7*, greater realism can be added by incorporating hidden surface removal, shading and shadow algorithms at appropriate stages of the operation

In contrast to this conventional approach, which starts with an object in the scene and tries to determine which pixels the object covers when projected onto the screen, ray tracing adopts the *reverse* methodology by starting with a pixel and trying to determine which object that pixel maps onto in the scene Thus, it could be said could be said that conventional approaches map objects onto pixels, while ray tracing maps pixels onto objects

In its simplest form, ray tracing involves casting an imaginary ray (representing a ray of light) through each screen pixel into the scene If the ray fails to strike any object in the scene, the pixel is given the background colour Otherwise, the colour of the pixel is determined from the characteristics of the nearest object struck by the ray, in accordance with the lighting model being used

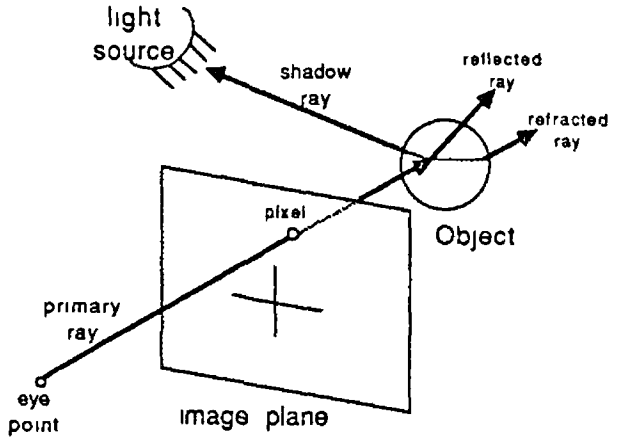
The generation of realistic graphics images is not however the only application of the technique of ray tracing since, the ray could equally well represent say, an x-ray, an acoustical path or the path of a light beam through an optical system For example, ray tracing has been used by Roberts [ROBE72] to predict acoustical ray paths in the atmosphere, by Maxwell [MAXW86] to calculate the radiation configuration factor between two surfaces, and by Higdon [HIGD74] to determine the path of a light beam through a system of reflecting and refracting optical elements, as an aid in the design of such systems

2.2 The Basic Algorithm

Fig 2 1 illustrates the basic idea behind ray tracing an image of some scene For each pixel, a ray is cast through the pixel and into the scene The first object struck by the ray while "tracing" along it is the visible one for that pixel The surface normal at the ray-surface intersection point is then calculated which, along with the position of the light source, is used to calculate the colour of the pixel The process can be subdivided into three distinct operations, ray generation, ray intersection and shading

RAY GENERATION The ray can be conveniently represented as a line in 3D space, usually defined in parametric form as a point (X_0, Y_0, Z_0), and a direction vector (D_x, D_y, D_z) Given this form, the points on the line are ordered and accessed via a parameter, t Each value of t gives rise to a point (X, Y, Z) on the line given by -

$$
\begin{aligned}
X &= X_0 + tD_x \\
Y &= Y_0 + tD_y \\
Z &= Z_0 + tD_z
\end{aligned} \tag{2 1}
$$



Tracing a ray through a scene of objects

figure 2 1

Positive, increasing values of t give points on the ray that are increasingly further from the point (X_0, Y_0, Z_0) in the direction of the ray, while decreasing negative values give points that are increasingly further from it in the opposite direction

For a parallel view defined by a direction vector (D_x, D_y, D_z), the equation of a ray through a pixel (X_v, Y_v) is defined by the point $(X_v, Y_v, 0)$ and the direction (D_x, D_y, D_z), *fig 2 2*

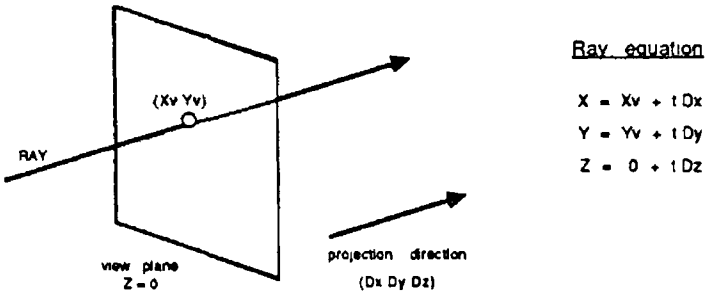


figure 2 2 Ray equation for a parallel view

Similarly, for a perspective view defined by a viewing point at (V_x, V_y, V_z), the equation of a ray is derived from the point $(X_v, Y_v, 0)$ and the direction $(X - V_x, Y - V_y, -V_z)$, *fig 2 3*

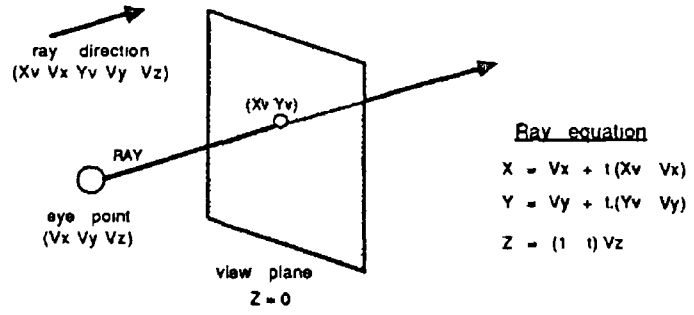


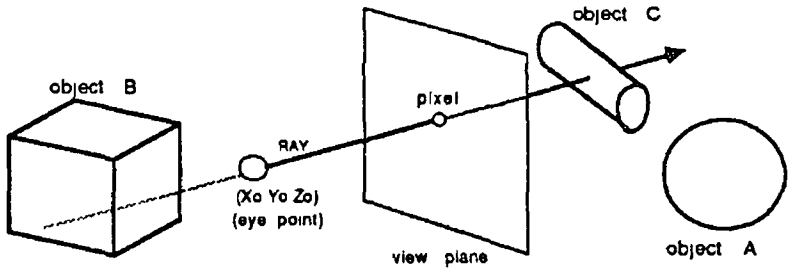
figure 2 3 Ray equation for a perspective view

Once in this form (a point and a direction) the ray is then passed to the ray intersection stage where the closest object of intersection with the ray (if any) is determined

RAY INTERSECTION Finding the closest object of intersection with a ray involves checking the ray for intersection with every object in the scene by determining if there is a value of t for which (X, Y, Z) in *equation 2 1* lies on the object In trying to find a solution for t, there are three possible outcomes, which are outlined in *table 2 1* below and illustrated in *fig 2 4* -

OUTCOME	INTERPRETATION	EXAMPLE
no solution for t	no intersection with object	object A in *fig*2 4
t is negative/zero	intersection behind/at the point (X_0, Y_0, Z_0)	object B in *fig*2 4
t is positive	intersection in ray direction	object C in *fig*2 4

TABLE 2 1



Three possible outcomes of a ray- object intersection test, represented by objects A B and C

figure 2 4

Having solved for t for each object in the scene, the object with the lowest positive value of t is the first object struck by the ray, since t is a measure of the distance of the point of intersection from the ray origin (X_0, Y_0, Z_0) Substituting this value for t into *equation 2 1* gives the actual point of intersection between the ray and the object, which can then be used to calculate the surface normal at that point for use in the shading calculation

The mathematics of the intersection test will depend on the object representation scheme in use To date, algorithms have been developed for a large variety of object representations such as polygonal objects [HECK84], algebraic surfaces [HANR83], parametric patches [KAJI83B], steiner patches [SEDE84], parametric surfaces [TOTH85], objects defined by sweeping a sphere [VANW84], superquadric solids [EDWA82], fractals [KAJI83A] & [BOUV85] and volume densities [KAJI84], as well as objects defined using a CSG (Constructive Solid Geometry) representation, [ROTH82] [YOUS86] and [ARNA87]

Since the determination of the closest object of intersection with a ray is based on a uniform test (a value for t in *equation 2 1*), regardless of object representation, different object representation schemes can be mixed in the same scene All that is required is that an appropriate intersection algorithm be incorporated into the raytracer for each different object representation scheme The ray intersector then proceeds through the list of objects in the scene calling the appropriate intersection algorithm for each object (which will return a value of t for the object) and selects the object with the smallest positive t value as outlined above The other components of the raytracer, the ray generator and the shader remain unaffected since the former merely fires rays into the scene while the latter simply requires a point of intersection, a surface normal and surface details such as colour, to determine the shade of a pixel

THE SHADER The shader makes use of the surface normal and the char-

acteristics of the intersected object (such as colour) to determine the colour and intensity of the pixel, in accordance with the lighting model being used If several light sources are in use, the shading calculation is performed separately for each one to determine its contribution to the overall intensity If required, the intensity of the light reaching the point of intersection can be attenuated in proportion to its distance from the light source However, if the ray did not intersect any object, or if all intersections were behind the the ray origin (X_0, Y_0, Z_0), (*i e* only negative values of t were found) the pixel is set to some background colour

2.3 Adding To The Algorithm

The ray tracing algorithm as outlined above will manage to produce screen images containing objects that have hidden surfaces removed and that are locally shaded (*i e* the shade of any point on the object depends solely on the orientation of the surface normal at that point with respect to the viewer and the light source, and is independent of the overall context of the object in the scene) A much greater degree of realism can be achieved however, by adding to the algorithm the capability to calculate shadows, to deal with reflecting and/or refracting objects and to perform antialiasing The first two are incorporated by generating what are called secondary rays, while the third calls for the generation of additional primary rays[1]

2.3.1 Shadows

A point on an object is said to lie in shadow with respect to a light source if the point can be seen from the viewpoint but not from the light source Having calculated the point of intersection of a primary ray with an object, it is possible to determine if the point lies in shadow by casting a ray from the point to the light source (if there is more than one source, a ray is cast from the intersection point to each one) If this ray intersects any opaque object in the scene, then the point lies in shadow with respect to the light source, otherwise it does not

In testing this "shadow" ray for intersection with objects it is possible to take advantage of the fact that we are not interested in finding the closest object of intersection with the ray, only if the ray strikes any opaque object Consequently, as soon as the ray intersects an object which is not transparent, no further objects

[1] A primary ray is one that originates at the viewpoint and passes through a pixel on the screen, while a secondary ray is one that originates from some point on an object in the scene

need be tested since the point then lies in shadow. If the ray strikes a transparent object however, the test must proceed, though the intensity of the light from the source can be reduced, if desired, to take account of the attenuating effect of the object on the light intensity.

In tracing the shadow ray back to the light source, the ray is not refracted as it passes through a transparent object (*section 2.3.2* discusses refraction), the reason being that it is not possible to directly calculate the equation of the ray from the light source which, when refracted will pass through the intersection point of object *A* in *fig 2.5*. Instead the shadow ray equation must be taken to be a straight line between the point and the light source, which as the figure shows, can sometimes give erroneous results.

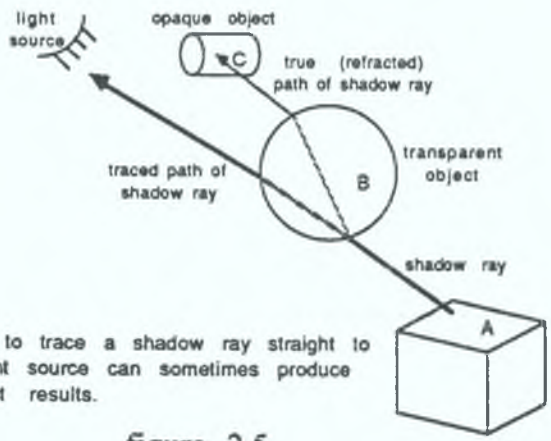


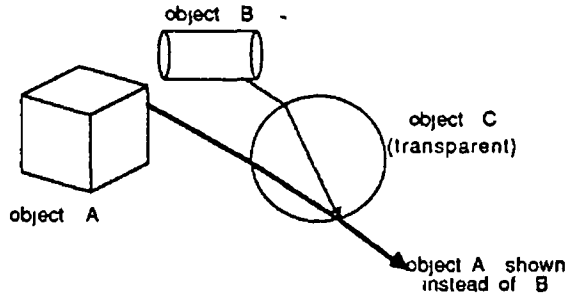
figure 2.5

2.3.2 Reflection and Refraction

Just as surfaces exhibit specular and diffuse *reflection* of light (*section 1.7.2*), so too can they exhibit specular and diffuse *transmittance*. Specular transmittance occurs in transparent materials such as glass, where light can pass through the material but is usually refracted, while diffuse transmittance occurs in translucent materials such as frosted glass where, although the light can pass through, it gets "scrambled" by the rough surface, with the result that objects seen through such a material are blurred.

Conventional hidden surface algorithms model transparent objects by ignoring refraction and shading them as a weighted sum of the individual shades calculated for the object itself and the object behind it. However, from *fig 2.6*

it can be seen that this can sometimes lead to the wrong object being shown through the transparent one



Failure to account for refraction means that object A instead of B is seen through the transparent object

figure 2 6

Turner Whitted [WHIT80] introduced a lighting model based on earlier ray tracing work by Appel [APPE68] that models refraction of light rather than attempt to simulate its effects The model also incorporates global illumination information in shading an object, *i e* it takes account of the effects of objects acting as secondary light sources and of objects being reflected in other objects

The model proposes that on striking an object a ray be split up into its specularly reflected and transmitted rays Each of these two component rays are then traced through the scene If either ray strikes another object, it in turn is subdivided into its two component rays, which are then traced In this way, a binary tree is recursively generated which contains a record of the light rays reflected from and refracted through other objects which contribute to the colour of the light reaching the viewer along the direction of the original primary ray, *fig 2 7*

In creating the tree, the ray intersection algorithm is called recursively until either all branches in the tree are terminated, or the tree reaches some predefined maximum depth The latter case is to safeguard against a situation where two objects may be aligned such that the tree has infinite depth Alternatively, the tree can be dynamically pruned by taking account of the attenuating effect of distance and of transparent objects on light, and stopping when it is such that the intensity has been reduced to a level where it is too low to make a notable contribution

Once created, the tree is passed to the shader which, starting with the leaf nodes calculates the contribution of each to the colour and intensity of the parent

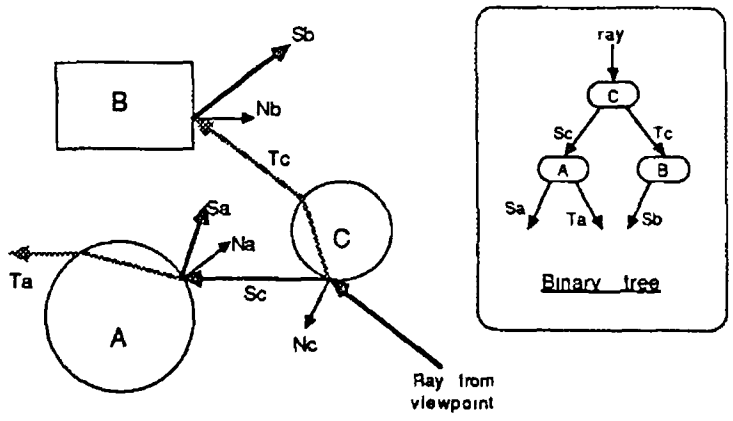


figure 2 7 Shade tree grown from a single primary ray

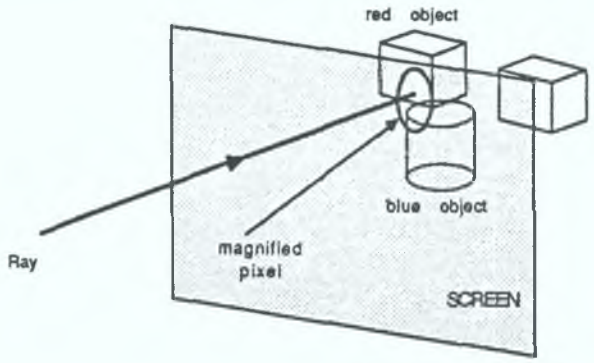
node, ending with the root node, which represents the colour of the pixel

2.3.3 AntiAliasing

Aliasing is a "noise" effect that manifests itself in graphics images as a result of attempting to display an object, which is continuous, on a screen which is not — it consists of a grid of points/pixels The problem of aliasing in computer shaded images is addressed by Crow in [CROW77B], and a comparison of antialiasing techniques (methods of removing/reducing aliasing effects) can be found in [CROW81] Ray tracing, being essentially a point sampling technique, is quite prone to the effects of aliasing but, as discussed below, the ray tracing algorithm can be adjusted to incorporate techniques that will reduce its damaging effect on picture quality

The Problem Ray tracing has a tendency to suffer from aliasing by virtue of the fact that it produces an image of a 3D model by sampling the model (through the use of rays) only at a specific number of points (the pixels on the screen) For each pixel, a ray is cast through the pixel into the scene to determine a colour for the pixel Part of the problem of aliasing, *fig 2 8*, lies in the fact that the ray, being represented as a mathematical line, has no thickness or area whereas the pixel does have a physical (though very small) area on the screen

Thus, a ray only allows us to sample *one* point of a pixel and there is no way of calculating what else is visible in the area of the pixel around this point Consequently, the pixel in *fig 2 8* would be shaded red by virtue of the fact that the ray, when fired through the centre of the pixel, strikes the red object This is despite the fact that the blue object covers as much of the pixel area as the



Tracing just a single ray per pixel can result in aliasing in the final image.

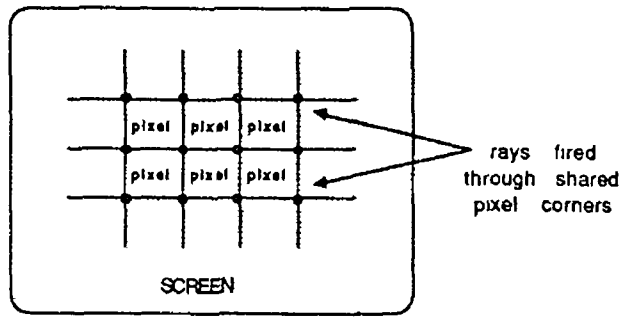
figure 2.8

red one. To avoid the aliasing produced by such a situation, the pixel should be shaded as the sum of the two object colours, weighted by the area of the pixel which each covers, a process known as *area sampling* a pixel. Despite the fact that ray tracing is essentially a point sampling technique, area sampling can be approximated by sampling the scene at more than one point on a pixel, as outlined below.

The Solution One solution is to use a technique known as *oversampling*, whereby more than one ray is cast through each pixel, allowing more than just a single point on the pixel to be sampled. The colour for the pixel is then calculated as the average colour of the values obtained for all the rays cast through the pixel. However, for a screen resolution of 500×500 pixels, casting N rays per pixel instead of one requires generating $(N-1) \times 250000$ extra rays which, while reducing the effects of aliasing increases the cost, computationally, of generating the image.

A more economical approach however, *fig 2.9*, is to approximate a pixel as a rectangular area on the screen and to cast rays through the corners of the pixels instead of the centre. Then, since every pixel shares several of its rays with neighboring pixels, only $(500+1) \times (500+1)$ rays are required to fire four rays through each pixel, which amounts to casting 1001 additional rays instead of 750,000.

If the intensities calculated at the four points have nearly equal values, then it is reasonable to assume that the average of the four values represents a good approximation to the intensity over the entire pixel region. If however the intensities differ by more than some threshold percentage, the pixel area is subdivided and new rays generated to probe the subregions of the pixel. This process is recursively repeated until a satisfactory degree of detail has been discovered for



Tracing rays through the corners of pixels means that adjacent pixels can share information

figure 2 9

the pixel, [WHIT80] The intensity of the pixel is then calculated as sum of the intensities calculated for each of the subregions, weighted by their area

Even with this latter approach, aliasing can still manifest itself in certain situations According to Mitchell [MITC87] however, non-uniform sampling yields aliasing that is less conspicuous to the viewer than that yielded by uniform sampling (such as the method above) He therefore presents an algorithm for fast generation of non-uniform sampling patterns that are optimal in some sense Another method, called distributed ray tracing and outlined by Cook, Porter and Carpenter [PORT84], also distributes the rays non-uniformly over the pixel, thereby overcoming the aliasing of uniform sampling As outlined in *section 2 5 1*, this technique also has the added advantage of distributing the rays in such a way as to be able to model such effects as motion blur, depth of field, penumbras and fuzzy reflections

In another approach by Amanatides, [AMAN84], the concept of Cone Tracing is outlined, whereby the definition of a ray is extended into a cone by including information on the spread angle and virtual origin of the ray Unlike a ray, a cone has thickness and consequently does not intersect an object at a point, but over an area, allowing area sampling of a pixel to be performed, (see *section 2 5 3* for a further discussion)

2.4 Speeding Things Up

The major drawback of ray tracing is that due to the large computational cost of generating images, rendering times are usually measured in hours rather than minutes According to Whitted [WHIT80] 75% of the time taken to render im-

ages is taken up with calculating the intersection of rays with objects This figure can rise to 95% and higher for complex scenes The reason for this is the sheer quantity of rays involved, together with the fact that each ray is tested for intersection with every object in the scene For example generating a 512×512 resolution image of a scene consisting of say 100 polygons, with just a single ray per pixel, requires 262144 rays and over 26 *million* ray polygon intersection tests If shadows, multiple light sources and reflection/refraction are also incorporated, the number of rays can go up by an order of magnitude Given that a scene of even moderate complexity would contain several thousand polygons, it is clear that there is a great need for optimization

One approach to the problem is to try to reduce the number of ray-object intersection tests, by selecting from the entire set of objects in the scene, a small subset of high probability candidates against which to test the ray for intersection This can be achieved through the use of any of the following techniques, which are discussed in the sections below -

[ ] Object bounding volumes

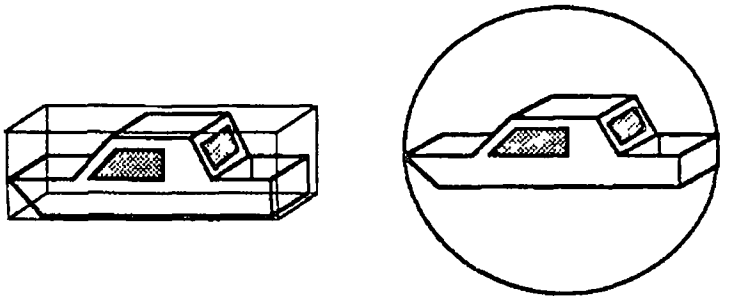
[ ] Space subdivision

[ ] Exploiting image coherence

2.4.1 Bounding Volumes

The use of *bounding volumes* aims at reducing the number of computationally expensive ray-object intersection calculations by enclosing each object in a volume, called a bounding volume (*e g* a sphere), which is less expensive to test for intersection than the object, *fig 2 10* Then, only if the ray intersects the bounding volume, is it tested for intersection with the object inside

Types To date, several different types of bounding volume have been used, with cylinders, spheres, and rectangular parallelepipes being the most common However, various other types have also been used For example, Kay [KAY84] bounds objects with parallelopipeds constructed of planes, Kajiya [KAJI83A] uses probabilistic extents to surround procedurally defined fractals and Bouville [BOUV85] compares ellipsoids, spherical triangles (the volume defined by the intersection of a sphere and a triangle) and triangular prisms as bounding volumes in tracing fractals



Object with rectangular and circular bounding volumes

figure 2 10

Characteristics In order to to get maximum benefit from bounding volumes, it is important that they should tightly enclose the object (*i e* there should be as little empty space between the object and the bounding volume as possible) since, the tighter the fit, the greater the percentage of rays that do not intersect the object will fail the bounding volume intersection test

However, the computational cost of testing a ray for intersection with a bounding volume is also another important factor in their use, since it would be pointless to have a tight fit if the intersection test were as costly as the object intersection test itself It is often the case that a more complex bounding volume will enclose an object more tightly than a simple one, but will be more expensive to test for intersection Since some objects in a scene will be more expensive to test for intersection than others, this may prove an acceptable trade off for those objects, as the relative cost of the bounding volume intersection test is less

The total cost function for an object is given by Weghorst [WEGH84] as -

$$C = b\,B + i\,I$$

where

C is the total cost

b is the number of times the bounding volume is tested for intersection

B is the cost of testing the bounding volume for intersection

i is the number of times the object is tested for intersection

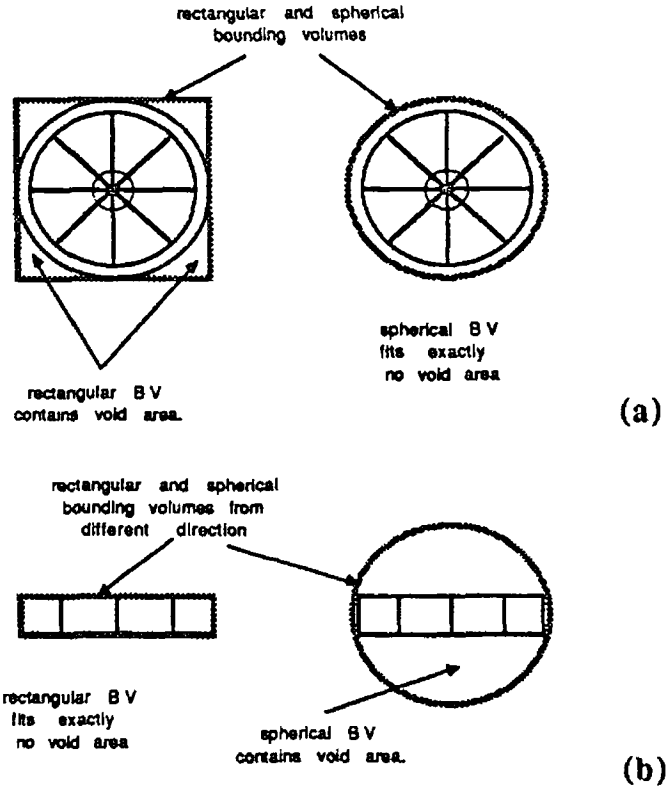
I is the cost of testing the object for intersection

The idea is to minimize this function for all objects in the scene For any given item with a given view, b and I are constant However, by selecting a less complex bounding volume, B can be reduced Unfortunately, this is likely to lead to an increase in i Similarly, increasing the complexity decreases i but is likely to lead to an increase in B Neither approach is guaranteed to decrease the overall cost function C, though there will usually be some optimal solution

The tightness of fit of a bounding volume is often a function of direction i e it will not enclose the object tightly on all sides, or, put another way, the *projected void area*[2] will vary for different directions The result of this is that a bounding volume which results in a small void area for one direction may not do so for all directions This can make the choice of the optimal bounding volume difficult at times, as can be seen from *fig 2 11a*, where the void area between the spherical bounding volume and the object (a wheel) is zero, while that of the rectangular one and the same object is not For a different direction however, *fig 2 11b*, the situation is reversed Thus, the sphere provides a better fitting bounding volume for rays coming from one direction, while the rectangular block provides a better fit for rays coming from a different direction

Selection As mentioned earlier, the two principle criteria of bounding volume selection are tightness of fit (which can be measured as the projected void area and can vary for different ray directions) and simplicity of intersection testing Unfortunately, the two are generally in conflict with each other in the sense that a gain in one is usually achieved at the expense of the other This conflict can make the optimal choice of bounding volume for a particular object quite difficult at times At one end of the scale, the selection can be based primarily on the tightness of fit (within the bounds of the intersection test expense not exceeding that of the object it encloses) For example, Kay and Kajiya [KAY84] have implemented a bounding volume that can be made to fit convex hulls arbitrarily tightly, at the expense of a more costly intersection test At the other end of the scale, the selection can be based mainly on the simplicity of the intersection test without worrying about a tight fit Such is the case in Whitted [WHIT80] who

[2]The projected void area for a particular direction is the difference in the projected areas of the bounding volume and the object when orthogonally projected onto a plane perpendicular to the direction in question



Wheel with rectangular and spherical bounding volumes from two different viewing angles (a) and (b)

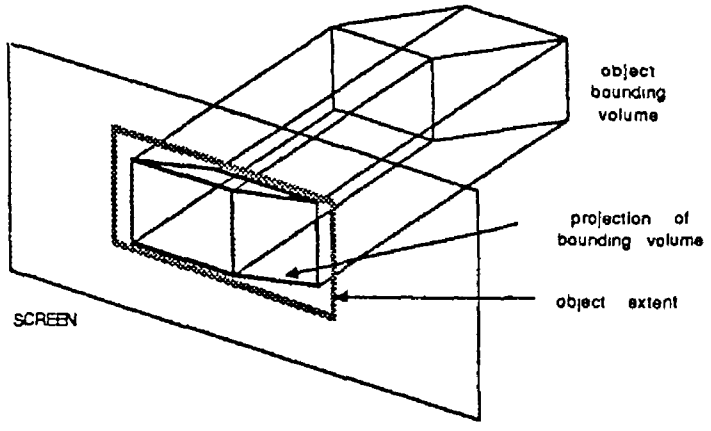
figure 2 11

uses spheres to bound all objects, since the complexity of the intersection test is relatively low and is uniform for all ray directions

Weghorst et al [WEGH84] however, adopts an intermediate approach The bounding volume for each object is optimally selected from a set of three differently shaped volumes, a sphere, a rectangular parallelepiped and a cylinder, each of which has associated with it a factor that is indicative of the relative complexity of its own intersection test compared to that of the other two (the sphere has the lowest factor and the cylinder the highest) For each object, a bounding volume of each type is generated that encloses the object, and the one with the least product of volume and complexity factor is selected An interactive program allows the selected bounding volume to be manually overridden to compensate for the fact that the cost of the object intersection test is not taken into account in the selection process Weghorst gives tabulated results of image generation times of several test images using these selected bounding volumes, comparing them with those taken to generate the same images using only spherical bounding volumes (times for other combinations of speedup techniques are also given) In

each case, the generation times for the selected bounding volumes are less than those using only the spherical bounding volumes, with the time saving varying from 49% to 10%, depending on the image

EXTENTS An *extent* could be loosely defined as a two dimensional bounding volume in screen space (as opposed to object space) For example, if all bounding volumes are projected onto the screen using the same viewing parameters as for the ray tracing, the "ray intersects bounding volume test" can be reduced to a point in rectangle test *i e* to test if a primary ray intersects an objects bounding volume, we simply check if the pixel spawning the ray lies inside the screen rectangle enclosing the projection of the bounding volume on the screen, *fig 2 12* Despite this less costly intersecrion test, extents have the limitation that they cannot be applied to secondary rays, since these rays are not constrained to pass through the screen However, they are relatively simple and inexpensive to implement and have been used successfully by Roth [ROTH82] in raytracing objects defined by a CSG (Constructive Solid Geometry) representation, *section 3 3*

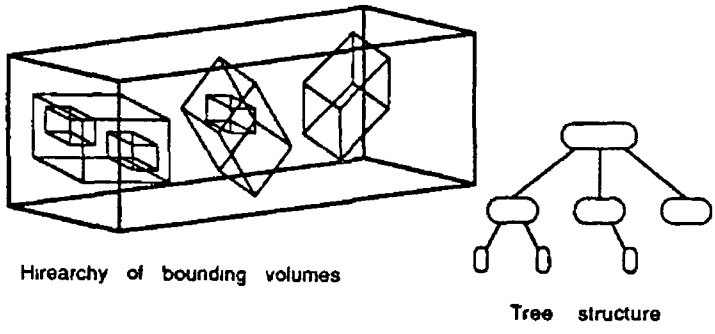


Calculation of an objects screen extent from its projected bounding volume

figure 2 12

HIERARCHY Having enclosed each object in a scene in a bounding volume of some sort, finding the object of closest intersection with a ray still involves having to test each bounding volume for intersection with the ray (and if it intersects, with the object inside) However, enclosing several bounding volumes inside a larger bounding volume means that the ray can first be tested for intersection with the outer bounding volume Then, if the ray misses this bounding volume, it does not have to be tested with any of the bounding volumes inside In turn, several of these outer bounding volumes can be enclosed in a still larger bounding volume and so on In this way, a *hierarchy* of bounding volumes can be built

up in the form of a tree, where the leaf nodes consist of the object bounding volumes and the intermediate nodes consist of bounding volumes that enclose the bounding volumes of of their respective subtrees, *fig 2 13* The root node then consists of a volume that encloses the entire scene and, finding the object of closest intersection with a ray involves descending the tree structure from the root node, recursively processing the subtrees of any node whose bounding volume is intersected At each level, those branches not intersected by the ray are pruned from the search Additionally, if an object is intersected, those branches of the tree whose bounding volumes lie behind the point of intersection can also be pruned, a process known as *dynamic tree pruning*



A bounding volume hierarchy and its tree structured representation

figure 2 13

In the limit, as advocated by Rubin and Whitted [RUBI80], the leaf nodes themselves can be bounding volumes, in which case the scene can be represented entirely by bounding volumes, with no other form of representation The bounding volumes used by Rubin consist of parallelepipeds orientated to minimize their size Such a structure allows any surface to be rendered since, in the limit, the bounding volumes make up a point representation of the object The visibility calculations then consist only of a search through the structure to determine the correspondence between the terminal level bounding volumes and the current pixel The main advantage of such a representation is that the viewing process has only a single operation (the search through the structure) and a single primitive type (a bounding volume), which allows the search to be highly optimized and makes it a suitable candidate for a hardware implementation

Extents can also be built up into a hierarchy Roth [ROTH82] for example uses a hierarchy of extents to enclose objects defined using a CSG representation, where the extents are embedded in the object definition tree itself and are used to prune non-intersected branches from the intersection test, *section 3 3 6*

The creation of a bounding volume hierarchy is a non-trivial operation, since overlapping of object bounding volumes in space should be minimal and the tree should be balanced and organized in such a way as to reflect the geometric distribution in space of the objects in the scene [JANS85] A poorly structured tree will probably take less effort to construct than a well structured one, but is also likely to result in a longer rendering time for the image For this reason, Rubin and Whitted [RUBI80] use a program that allows a user to interactively built up a hierarchy for a scene, though they also make several proposals for automating the process Weghorst [WEGH84] also uses a hierarchy, defined by the user during modelling, and produces tabulated generation times for several test images using various combinations of spherical bounding volumes, selected bounding volumes, hierarchy, and a visible surface preprocess (see *section 2 4 5*) In each case, the use of a hierarchical structure reduces the time required to render an image, with improvements varying from 14% to 21%, depending on the image However, the structure of manually generated hierarchies can sometimes be poor, hence Goldsmith and Salmon [GOLD86] have proposed a method for the automatic creation of such hierarchies

2.4.2 Space Subdivision

In contrast to the above bounding volume approach to reducing ray object intersection tests, which is an object orientated approach, the *space subdivision* approach, as the name might suggest, is space orientated The general idea is that the 3D space in which the objects are contained is divided into a number of 3D cells Associated with each cell is a list of all objects either completely or partly contained in the cell Then, given that a ray enters a particular cell, only the objects contained in the object list for that cell need to be tested for intersection Assuming that the cells are checked in the order in which the ray will enter them, the search can be terminated as soon as the closest object of intersection has been found for the first cell in which an intersection occurs In contrast to a hierarchical bounding volume scheme, which functions better when objects can be grouped into close clusters, the above scheme is better suited to a situation where all objects are uniformly distributed over the entire object space

Cells in a space subdivision scheme differ from bounding volumes in that a cell may sometimes contain only part of an object, which can lead to a particular object being associated with more than one cell In contrast to this, an object is never only partly enclosed by a bounding volume Another significant difference is the fact that the total sum of the volumes of all of the cells represents the entire 3D space enclosing the scene *without* duplication, whereas, the sum of the

volumes of all of the object enclosing bounding volumes does not

In subdividing the object space into cells, several different techniques can be used, the principle ones of which are listed below in order of increasing complexity and illustrated in *fig 2 14*

[ ] Uniform subdivision

[ ] fixed adaptive subdivision

[ ] unequal adaptive subdivision

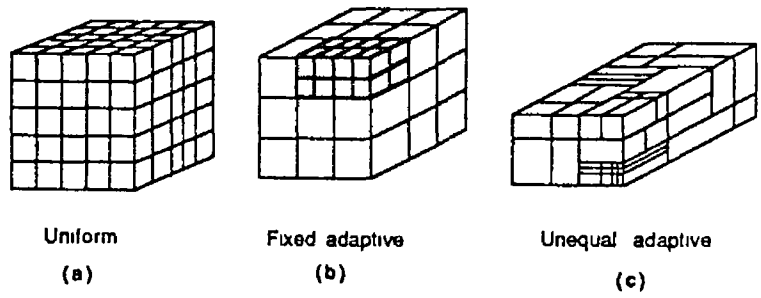


figure 2 14 Space subdivision techniques

UNIFORM SUBDIVISION This is the simplest type of cell structure, in which the space is divided into a three dimensional rectangular grid of cells of equal size, *fig 2 14a*, and is best suited to a situation where the objects are fairly uniformly scattered through the scene A ray can be propagated from one cell to the next by extending the Digital Differential Analyzer [NEWM79] (a method for generating lines on a 2D raster grid) to three dimensions, a technique used by Fujimoto [FUJI86]

An important parameter of the division is the number of cells to use If too few are used, the technique degenerates to the basic tracing algorithm since the size of the cells will be large, as will the number of objects associated with each If on the other hand, too many cells are used then too much time will be wasted propagating the ray through a large number of mostly empty cells Measurements by Fujimoto indicate that for most scenes, the number of ray intersection calculations decreases quadratically as the number of cells in a uniform subdivision increases Since the time taken to propagate the rays increases linearly, optimal performance should occur where the sum of these two curves is min-

imised However, this optimal number is also heavily dependent on the extent and distribution of objects within the scene

FIXED ADAPTIVE SUBDIVISION The uniform subdivision technique has the disadvantage that in a situation where the objects in a scene are unequally distributed throughout the scene, some cells will have a large number of objects associated with them, resulting in a costly ray propagation through those cells Fixed adaptive subdivision, *fig 2 14b*, attempts to overcome this drawback by defining a maximum number of objects that can be contained in a single cell If this maximum is exceeded for any cell, the cell is subdivided into a fixed number of smaller equally sized cells, and the objects redistributed among them The cost of this optimization however, is that it is more costly to propagate a ray from one cell to the next for this scheme than for uniform division

A common method of implementing a fixed adaptive subdivsion scheme is to use an octree structure to represent the cells since, an obvious way to subdivide a cell is to divide it equally in two along each of the X, Y and Z axes, giving rise to eight equal subcells The ray can then be then be propagated from one cell to the next by by traversing the octree structure, with the object data being accessed via the leaves of the tree An alternative means of propagation is by a directory index method whereby, an address computation is performed on the basis of the coordinates of a particular point and followed by a lookup in a directory table called a *spatial index*

UNEQUAL ADAPTIVE SUBDIVISION In the fixed adaptive subdivision scheme outlined above, a cell which exceeds the allowed maximum number of objects, is subdivided by placing a partition in the middle of each of the X, Y and Z axes, giving rise to eight equal subcells The unequal adaptive subdivision scheme however provides a more flexible subdivision, by allowing multiple partitions at arbitrary positions along any of the three axes, *fig 2 14c* Due to this flexibility of partitioning, less space is required to represent an object, since the partitions can be placed so as to minimize the number of subcells This is achieved however at the cost of an even more expensive ray propagation scheme than for fixed adaptive subdivision

Dippe and Swensen [DIPP84] have implemented such a subdivision scheme by using a fixed number of arbitrarily shaped tetrahedral cells, each adapted in size to contain an equal distribution of objects over all of the cells This structure is then used to distribute the ray tracing load over a fixed number of parallel processors A similar scheme is also employed by Nemoto [NEMO86], using orthogonal parallelepipeds for cells

2.4.3 Coherence

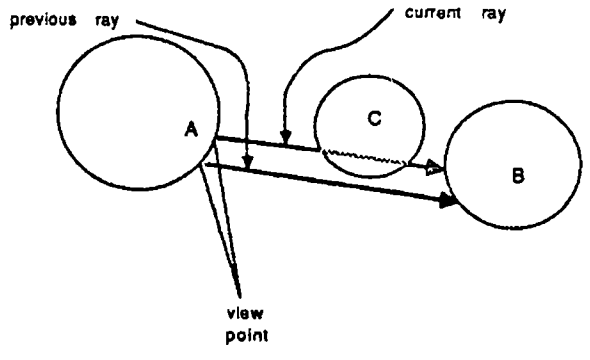
Another approach to reducing the time taken to ray trace a scene is to exploit the similarity of the intersection trees generated by successive rays Such similarity, or *coherence* (the extent to which a scene or a picture of it is locally constant) has often been used in traditional rendering algorithms With regard to ray tracing, it has been noted by Heckbert [HECK84] that in many scenes, groups of rays follow virtually the same path from the eye Therefore, instead of discarding the ray intersection tree for a pixel (*section 2 3 2*) as soon as the shader has used it to calculate the intensity of the pixel, the tree is retained and used to predict the ray paths for the next pixel

Verroust [VERR85] for example, takes into account the coherence of an image to reduce the number of rays fired as part of a hidden surface removal algorithm that produces wire frame pictures of CSG models In ray tracing parametric surface patches, Joy [JOY86] also utilizes coherence by using numerical information from adjoining rays as initial approximations to a quasi-Newton iteration employed to solve ray-surface intersections As a result, a significant number of ray-surface intersections can be found using much fewer iterations, resulting in a significant computational saving PORTRAY, an image synthesis system that uses ray tracing to produce realistic images [PEAC86], also employs coherence by means of a technique of exploiting bounding volume coherence

Speer [SPEE85] examines the theoretical and empirical performance of a coherent ray tracing algorithm that exploits the similarity of the intersection trees generated by successive rays However, his results show that the overhead of ensuring the validity of ray-object intersections prevents any significant computational savings, even in a scene where there is a large degree of coherence The need for such validation can be seen from *fig 2 15*, where the current ray intersects the closest object intersected by the previous ray, but also intersects a closer intervening object missed by the previous ray As a result of such cases, when a ray intersects the same object as a previous ray, it cannot be safely assumed that this is also the closest object of intersection for the ray Consequently, some of the benefits of coherence are lost in checking to see if a closer object is intersected, that was not intersected by the previous ray

2.4.4 Parallel Algorithms

As mentioned earlier, the majority of the time taken to ray trace most images is taken up with ray-object intersection testing Most of the optimizing techniques



Current ray intersects object hit by previous ray but also strikes an intervening object

figure 2 15

outlined above address the problem by trying to reduce the number of objects against which a ray must be tested Another approach however, is to realize that most of these calculations can be carried out independently of each other and hence can be performed *simultaneously* on different processors

Perhaps the simplest and most extravagant approach to parallel ray tracing would be to independently run the ray tracer on several machines, with each machine calculating a different part of the final screen image The image files generated on each machine could then be collected and amalgamated to form a final image The reduction in the time taken to ray trace an image would then be directly proportional to the number of machines available This approach however, while simple and effective, is extremely wasteful of the available resources, since each machine must have its own copy of the ray tracer and scene, resulting in multiple duplication of information

Another approach is to perform the ray-object intersection tests in parallel by dividing the list of objects against which a ray must be tested among the available processors This is the approach used by Nemoto who presents an adaptive subdivision algorithm for fast ray tracing which has been implemented on a parallel architecture consisting of a three dimensional computer array, [NEMO86] The algorithm involves dividing the object space into as many subregions as there are computers and adaptively sliding the boundary surfaces of the subregions so that processor loads are uniformly distributed, thereby overcoming the problem of load concentration on a particular processor

A different parallel algorithm, developed and implemented by Deguchi on a distributed parallel processing system, uses a hierarchical tree structured architecture instead of the 3D array architecture used by Nemoto The hierarchical tree-structured image generation system and its parallel processing mechanisms,

such as data transfer and hierarchical load distribution schemes are outlined in [DEGU86].

Cleary [CLEA83] outlines a multiprocessor algorithm for ray tracing and analyses its performance for a cubic and a square array of processors with only local communication between near neighbors. Theoretical expressions for the speedup of the system for both configurations are supported by simulations for several scenes and indicate that a square array of processors generally performs better than a cubic one.

In contrast to the above algorithms, which use multiple processors to achieve parallelism, Plunket [PLUN85] implements a vectorized ray tracing algorithm that takes advantage of the full power of the CYBER 205 supercomputer to trace rays in parallel on a single processor. Instead of tracing each ray immediately, the ray is placed in a ray queue. When this queue becomes full, the vector processor of the CYBER 205 fires all rays in the queue at once. The ray tracing program then goes back and uses the results where they were originally requested. This retooling of the algorithm results in significant speed increases in ray tracing times.

2.4.5 Other Speedups

Hardware In addition to the various strategies outlined above, several other "miscellaneous" approaches that do not easily fit into any of the above categories have also been developed. One such approach is that by Pulleyblank [PULL87], who examines the feasibility of a VLSI chip for calculating the intersection of a ray with a bicubic patch to a specified level of accuracy. Estimates indicate that such a chip could compute ray-patch intersections at a rate of one every $15ms$. Images drawn using a software version of the intersection algorithm are also presented.

Preprocess An optimization that does not rely on hardware is that by Weghorst [WEGH84] who uses a visible surface *preprocess* to the raytracing algorithm to reduce rendering time. The preprocess involves projecting all objects onto the image plane (using the same viewing parameters as for the ray tracing) and creating an object list for each pixel, containing all objects that projected onto all or part of the pixel. Finding the closest object of intersection for a primary ray (one that passes through a pixel) simply involves testing the objects in the associated list for the pixel concerned. The idea can be taken a stage further by applying a conventional hidden surface algorithm as part of the preprocess (Weghorst uses

a modification of the z-buffer algorithm, *section 1 7 1*) to determine, and store in an item buffer, the closest object for each pixel The ray tracing process can then replace the ray-object intersection test for primary rays with an index into the item buffer, while using the usual ray tracing method for secondary rays The technique is aimed at reducing the cost of the intersection for primary rays only, and so may not work as well for very reflective/refractive scenes where the proportion of secondary rays is higher

Stepwise Refinement A different approach by Bronsvoort [BRON84] uses a method of stepwise refinement of the image by subdivision and aims at reducing the cost of rendering an image by reducing the number of pixels whose intensities have to be explicitly calculated The algorithm is based on ideas from Roth [ROTH82] for sparce sampling of images and from Whitted [WHIT80] for antialiasing images It starts by dividing the screen into squares consisting of groups of pixels (*e g* 8×8 pixels) Rays are then traced through pixels in the lower corners of the squares in the usual manner, with all other pixels in a square being set to the intensity calculated for the corner pixel The image obtained is a coarse approximation to the final image The image is then refined by subdividing the squares into 4 equal subsquares and repeating the process for selected subsquares, depending on whether or not the intensity differences with surrounding ones are above some threshold value This subdivision process is repeated until a final user specified resolution is reached If required the process can be continued to sub-pixel level, resulting in an antialiased image The image is thus stepwise refined as the user watches on the display

Depending on the coherence of the image, the number of pixels whose intensities have to be explicitly calculated with a call to the ray tracing procedure can be significantly less than the total number of pixels, resulting in a computational saving Additional savings can be obtained by dynamically increasing the initial threshold value as the resolution of the image is stepwise refined, so that at higher resolutions, only areas with a large variance are refined

The optimization has the disadvantage however, that silvers may occasionally be lost from the final image, due to the fact that many pixel intensities are not explicitly calculated, but based on the values of neighboring pixels The performance of the algorithm in recovering such detail from a scene is dependent on the initial square size and threshold values, which are specified by the user Thus, for any image, the user can favor generation time over quality (or vice versa) by appropriately selecting these values

2.5 Other Ray Tracing Algorithms

To date, several different variations of the basic raytracing algorithm outlined in the previous sections have been implemented These include distributed ray tracing [PORT84], which has been used to model effects such as fuzzy shadows, motion blur and depth of field, as well as variations that trace more than one ray at a time, such as beam tracing [HECK84], cone tracing [AMAN84] and pencil tracing [SHIN87] Each of these variations are briefly outlined below

2.5.1 Distributed Ray Tracing

One of the limitations of conventional ray tracing is that ray directions are determined precisely from geometry, which results in sharp shadows and sharp reflections/refractions in the final image However, by *distributing* the directions of the rays according to the analytic function which they sample, ray tracing can incorporate fuzzy phenomena, providing correct and easy solutions to previously unsolved or partially problems such as motion blur, depth of field, penumbras, translucency and fuzzy reflections This form of ray tracing is known as distributed ray tracing, a phrase coined by Porter, Cook and Carpenter [PORT84]

The analytic function will vary depending on which of the effects is being modelled For example,

- sampling the reflected ray according to the specular distribution function produces gloss (blurred reflection)
- sampling the transmitted ray produces translucency (blurred transparency)
- sampling the solid angle of the light sources produces penumbras
- sampling the camera lens area produces depth of field
- sampling in time produces motion blur

For example, distributed ray tracing produces penumbras (fuzzy shadows)

by distributing illumination rays according to an illumination function, L, rather than toward a single light direction. Similarly, fuzzy or hazy reflections are produced by distributing reflected rays according to a reflectance function R, rather than in a single mirror direction. The shade to be displayed at any pixel is then a weighted integral of L and R. However, since this integral may be too complex to solve analytically, its value is approximated by firing several rays through the pixel, distributed so as to sample the integral at various points. Lee [LEE85] derives a relationship between the number of sample rays fired and the quality of the estimate of the integral.

While this distributed form of ray tracing requires that several rays be fired for each pixel, it is argued by Porter [PORT84] that the expense is not much greater than the oversampling required for antialiasing images (*section 2.3.3*), and that distributing the rays instead offers substantial benefits at little additional cost.

2.5.2 Beam Tracing

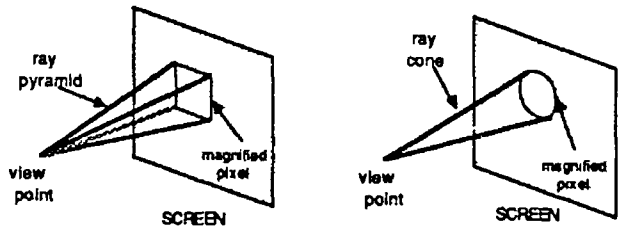
In conventional ray tracing, infinitesimally thin rays of light are traced through the scene. However, in a different approach used by Heckbert and Hanrahan [HECK84], *areas* are swept through a scene to form beams, hence the term *beam tracing*. The scenes used by the authors consist of planar polygons since, unlike the general case of a beam reflecting from a curved surface, the reflected beam from a planar one can be approximated by a pyramidal cone, which reduces the complexity of the calculations.

The algorithm is similar in principle to a technique developed by Dadoun, Kirkpatrick and Walsh [DADO82] to trace sound beams from audio sources to a receiver, and can take advantage of the coherence of polygonal scenes by tracing coherent rays (rays that follow similar paths) in parallel as a single beam.

2.5.3 Cone/Pencil Tracing

The problem of conventional ray tracing is that, being a point sampling technique, it is prone to aliasing. However, as outlined in *section 2.3.3*, the effects of aliasing can be reduced by sampling a pixel at more than one point. Unfortunately, tracing additional rays for each pixel adds to the already high computational cost of ray tracing an image. One way around this problem is to incorporate the idea that a pixel represents an area, into the definition of a ray. A ray then becomes a

pyramid, *fig 2 16*, with the apex at the eye and the base defined by the four planes that cut the border of the pixel The intersection calculation between such a ray an an object, in addition to determining if there is an intersection, calculates the area of intersection with the ray This information can then be used as a basis for performing simple area antialiasing



A ray pyramid and its approximation as a cone

figure 2 16

Given this extended definition of a ray, only a single ray need be cast for each pixel However, the intersection calculation between an object and such a ray can be quite complex In addition, as the ray is reflected of refracted by a curved surface, it can become very distorted, furthering the complexity of the intersection calculations Amanatides [AMAN84] addresses this problem by approximating the pyramid as a cone Although the intersection calculations can still be quite complex, the advantage of such an approximation is that, when reflected or refracted, a cone will still represent a good approximation to the reflected/refracted components of the original cone

In addition to providing a better means of antialiasing, cone tracing can also be used to calculate fuzzy shadows and dull reflections, as well provide as a means of calculating the correct level of detail in a texture map Shinya, Takahashi and Naito [SHIN87] describe a similar approach, called pencil tracing which uses paraxial approximation theory to trace a pencil of rays (rays that are near to a given axial ray are called paraxial and said to form a pencil), and describes an error analysis method to ensure its accuracy

2.5.4 Other Variations

Radiosity Another variation used by Wallace, Cohen and Greenberg [WALL87] uses a two pass solution to the rendering equation which is a synthesis of radiosity

and ray tracing methods As mentioned in *section 1 7 2*, the intensity of light reaching a viewer is composed of diffusely reflected/transmitted light and specularly reflected/transmitted light In most ray tracing applications however, the diffusely reflected/transmitted contribution from other surfaces in a scene is too costly to accurately model and is therefore usually approximated by an ambient term The specularly reflected/transmitted component on the other hand, can be approximated using Whitted's lighting model [WHIT80]

The radiosity method on the other hand, provides a comprehensive method of calculating object to object diffuse reflections within complex environments containing hidden surfaces and shadows [COHE85] [RUSH86] [SHAO88], but does not as easily handle the specularly reflected/transmitted component — a problem addressed by Rushmeier, [RUSH86] In addition, it has the advantage that the calculations are independent of the viewpoint, so unlike ray tracing, the image does not have to be entirely recalculated if the view-point is changed

Combining the two methods should therefore give a more accurate model of the lighting effects within a scene and hence a greater degree of realism in the final image The method employed by Wallace involves two passes The first is view independent and based on the radiosity algorithm, with extensions to include the effects of diffuse transmission The second, which is view dependent and based on an alternative-to distributed ray tracing, is used to sample the intensities contributing to the specularly reflected or transmitted intensity

Interreflection Rubinstein and Clear [WARD88] outline a raytracing algorithm that incorporates diffuse interreflection between surfaces with both diffuse and specular components A Monte Carlo technique is employed to calculate indirect contributions to illuminance at various locations These indirect illuminance values are then averaged over surfaces and used in place of the constant "ambient" lighting term

Ray Classification This approach to ray tracing, by Arvo and Kirk [ARVO87], reduces the number of ray-object and ray-bounds intersection calculations by means of 5-dimensional space subdivision Collections of rays originating from a common 3D rectangular volume and directed through a 2D solid angle are represented as hypercubes in 5-space A 5D volume enclosing the ray space is then dynamically subdivided into hypercubes Associated with each hypercube is a set of objects which are candidates for intersection Rays are then classified into unique hypercubes and tested for intersection with the associated candidate set

Chapter 3

Ray Tracing and CSG

While *chapter 2* discussed ray tracing in a general context, outlining the general algorithm, with its different enhancements, variations and optimization techniques, the discussion did not focus in any great detail on any one object representation scheme This chapter discusses ray tracing in the specific context of solid modelling, or more precisely, in the context of a Constructive Solid Geometry (CSG) representation scheme and provides a background setting for the research discussion in *chapter 4*

3.1 Solid Modelling

Solid modelling was born of the need for complete and accessible object-geometry information, such as that required for the integration of the design, simulation and manufacture phases of engineering products (*e g* an engine part, or even a complete engine) Such integration requires that any object-geometry information (*e g* volume, weight, centre of gravity *etc*) required for the simulation of the operation of the product, as well as for the manufacturing process (*e g* determination of machine tool paths), can be extracted from the original object representation interactively built up during the design stage

Wire frame, and even polygon representation schemes (*section 1 6 2*), are inherently ambiguous and consequently, important geometric properties of the objects they attempt to represent cannot be calculated On the other hand, solid modelling systems provide unambiguous and informationally complete representations of rigid solid objects (see [REQU80], [VOEL77] for an introduction)

Some of the more common schemes in use to date are outlined below -

Primitive Instancing This scheme provides families of basic objects such as blocks, spheres, cylinders and cones An object is then represented by its family name and parameters specifying its dimensions, orientation and position Such a scheme has the advantage of being concise, simple and effective, but is limited in the range of objects it can represent

Spatial Enumeration In this scheme, an object is represented by a list of fixed-sized cubes occupied by the object The smaller the size of the cubes, the greater the accuracy with which a given object can be represented but, the larger the storage requirement to represent the object However, the use of an octree structure, where the object is represented as a hierarchical collection of variable sized cubes, can reduce the storage requirements and speed up object processing

Sweep Operation A sweep operational scheme represents an object by means of a primitive object and a trajectory path The object is defined as the volume swept out by moving the primitive along the path Translational and rotational sweeps, used in conjunction with a spherical or cylinderical primitive are among the most commonly used

Constructive Solid Geometry As outlined in *section 1 6 2*, a CSG scheme represents an object in terms of compositions of primitive solids that are combined using boolean set operators (union, intersection and difference) The primitives usually used are the block, sphere, cylinder, cone and torus A CSG scheme, combined with a primitive instancing scheme to represent primitive objects, provides a very elegant and efficient method for representing solid objects

3.2 An Introduction to CSG

Constructive solid geometry is becoming the method of choice for a wide range of applications of engineering design This is in part due to the fact that the way in which objects are built up using a CSG system, namely through boolean operations on primitive and intermediate solids, reflects the way in which many engineering products are actually manufactured For example, the CSG union and difference operators are the equivalent of the physical operations of welding and cutting respectively In addition to this, CSG can be used to provide a means of representing solids that inherently describes their properties as solids, allowing the extraction of information such as contained volumes, masses, material costs, and on a more complex scale, changes of shape due to distorting forces and

machine tool paths for manufacturing purposes

Descriptions of CSG systems are given by Boyse & Gilchrist [BOYS82], who describe GMSolid, an interactive modeller for the design and analysis of solids, and by Brown [BROW82], who gives a technical summary of a system called PADL-2 Requicha & Voelcker [REQU82] present a more general discussion of the area however, outlining an historical summary and contemporary assessment of solid modelling, while Myers [MYER82] views the area from an industrial perspective

3.2.1 CSG Representation

As outlined in *section 1 6 2*, an object is described in CSG as combinations of union, difference and intersection operations on primitive and intermediate solids, *fig 3 1* The two most common structures for representing objects defined in such a way are a binary tree and a directed acyclic graph (DAG), *fig 3 2* In the binary tree structure, the leaf nodes represent primitive solids and intermediate nodes represent the intermediate solids formed by applying the specified operator to the left and right subnodes, with the root node representing the final solid In the DAG structure, each vertex of the graph represents either a primitive solid or an intermediate solid, with a specified vertex representing the final solid The DAG structure can, in some cases be more compact than the binary tree one, by virtue of the fact that the same sub-object can be used many times in a description without duplication by having several vertices point to it In the binary structure however, a sub-object can be pointed to by only one node (the parent one), which means having to duplicate the sub-object

However, both binary tree and DAG structures provide only a means of describing an object In order to render an image of the object, or to extract useful information from the structure, most systems convert the structure into a more conventional description such as a polygon mesh or boundary patch representation GMSolid (see [BOYS82]) is one such example Roth however, [ROTH82], developed a ray tracing technique for rendering objects directly from a binary tree structure, without the need to convert to a different representation While there has since been several other algorithms developed that can do likewise [YOUS86] [ARNA87], including some based on Roth's work, [BRON84] [GERV86], all are based on a ray tracing approach, which remains to date the only method of rendering an image directly from a CSG binary tree or DAG structure

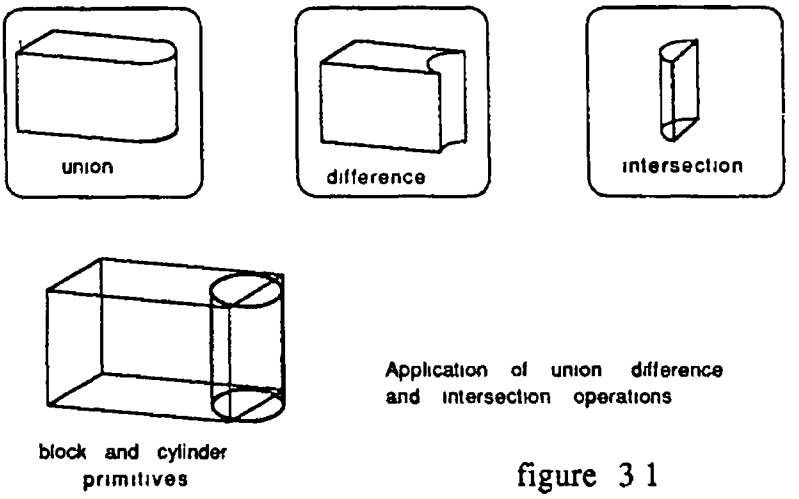
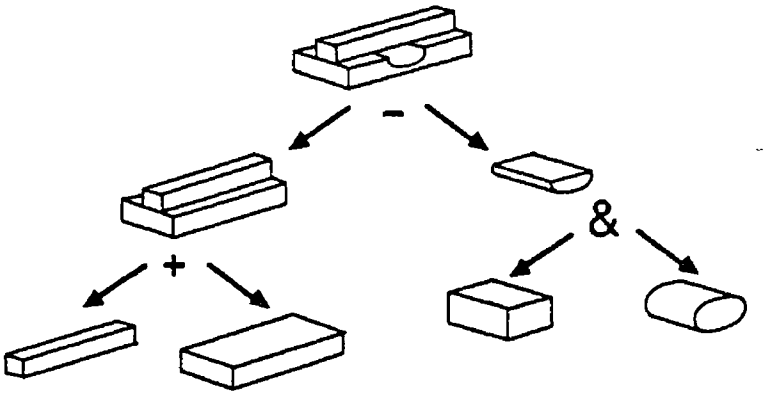
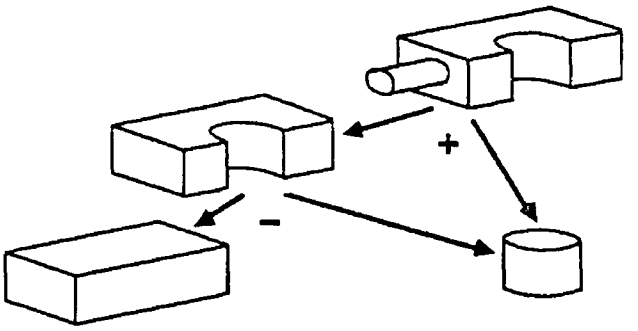


figure 3 1



(a) A compound object represented by a binary tree



(b) A compound object represented by a DAG The same cylinder is subtracted and subsequently with a different size and orientation

figure 3 2 CSG and DAG representstions of solids

3.3 Roths CSG Ray Tracing Algorithm

The general idea behind the technique developed by Roth is essentially the same as the ray tracing algorithm outlined in *section 2 2* in that rays, represented as line equations in parametric form, are cast through each screen pixel in order to determine the closest object struck by the ray and subsequently, a colour for the pixel The difficulty however hes in extracting from the binary tree representation of an object, the point of intersection between a ray and the object

In the system developed by Roth, this task is performed by RAYCAST, a procedure whose input is a ray and whose output is information about how the ray intersects the scene This output takes the form of two hsts, a list of t values (see *section 2 2, equation 2 1*) that specify the points at which the ray enters and exits the solid as it "passes through" it, and a list of surface pointers that point to the corresponding surfaces through which the ray passes, *fig 3 3*

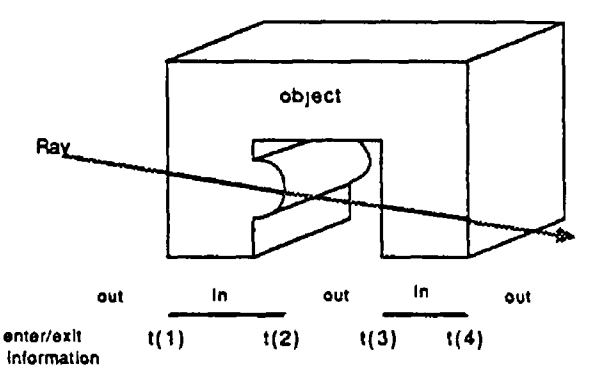


figure 3 3 In/Out classification of a ray by RAYCAST

3.3.1 Three algorithms in one

Given the information contained in the two lists, the algorithm can be modified to produce -

[ ] A line drawing of a solid

[ ] A shaded image of a solid

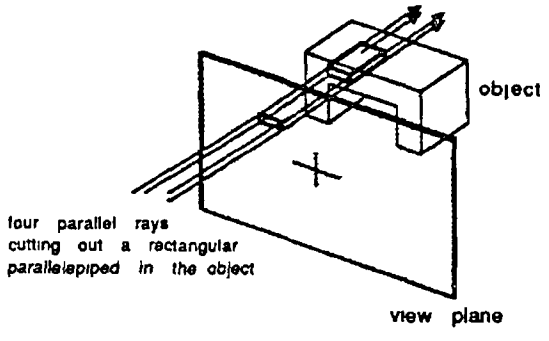
[ ] Calculate the volume of a solid

LINE DRAWING In producing a line drawing of a solid, we are only interseted in knowing if the surface struck by the ray for a given pixel is different to that struck by the ray for the pixel immediately to the left or above it We do not therefore need to know the exact point at which the ray intersects the surface Consequently, in producing a visible edge drawing of a solid, the algorithm uses only the surface list output by RAYCAST This firstly involves comparing the nearest surface struck by a ray cast through pixel (x, y) (the first pointer in the surface list) with that returned for pixel $(x - 1, y)$ If the surface is a different one, a vertical line one pixel long is displayed at $(x - 0\ 5, y)$ The pointer is then compared with that returned for pixel $(x, y - 1)$ If it is different, a horizontal line one pixel long is displayed at $(x, y - 0\ 5)$ In order for these comparisons to take place, a record must be kept of the pointers returned for all pixels processed so far on the current line, as well as for all pixels from the previous line

SHADED IMAGE The shaded image algorithm makes use of both the parameter list and the surface list output by RAYCAST It uses the first value in the parameter list to calculate the exact point of intersection of the ray with the closest surface struck (the first surface pointer in the surface list) in order to calculate the surface normal at that point This information is then used in the shading calculation to determine a colour for the pixel (*section 1 7 2*)

VOLUME CALAULATION The volume of a solid can be approximated by specifying a parallel view (*section 1 5 2*) so that all traced rays are parallel to a given direction The traced rays then divide the solid into rectangular parallelepipeds whose volumes can be individually calculated and summed to approximate the total volume, *fig 3 4* Two of the dimensions of each parallelepiped are determined by the horizontal and vertical spacing of the rays on the screen and are consequently known and the same for each parallelepiped The third dimension, the total length of the parallelepiped contained by the solid can be calculated from the in/out parameter list Given that the horizontal and vertical spacing of the pixels is H and V respectively, the volume for each ray is then calculated as

$$volume = H \times V \times (t_2 - t_1 + t_4 - t_3 + \quad t_n - t_{n-1}) \times L \qquad (3\ 1)$$



The volume of an object can be approximated by summing the volumes of the rectangular parallelepipeds cut out by parallel rays

figure 3 4

where L is the length of the direction vector of the ray which, if specified as a unit vector, can be omitted from the calculation The error of approximation can be made arbitrarily small by using larger quantities of smaller parallelepipeds, but this is at the expense of a having to generate more rays, increasing the cost of the calculation

Central to all three variations of the algorithm is the means by which RAY-CAST calculates the list of in/out intersection points Before moving on to this discussion in *sections 3 3 3 and 3 3 4*, a description of the primitive solids and coordinate systems used is outlined in the next section Then, having covered both of these areas, the remainder of the discussion focuses on the computational cost of rendering an image using the algorithm, and ways in which it can be reduced

3.3.2 Primitives And Coordinate Systems

As outlined earlier, an object is defined in CSG in terms of boolean operations performed on a set of primitive solid types and can be conveniently represented as a binary tree However, regardless of the complexity of the final solid (represented by the root of the tree) a ray must always enter and leave the solid through a surface of one of the primitive solids from which it is composed Consequently, it is important that ray-primitive intersection tests be simplified as much as possible It is for this reason that several different coordinate systems are employed in Roths ray tracing system, namely the screen, primitive and global (or world) coordinate systems The user of the ray tracing system however, should be aware of only the world coordinate system

The screen coordinate system, as outlined in *section 1 4 1*, is a 2D integer system used for referencing screen pixels The primitive coordinate systems stem from the fact that each of the primitive solid types is defined in its own local 3D coordinate system, and can be transformed into the world coordinate system in any size, orientation and position by applying the appropriate 3D matrix transformations outlined in *section 1 5 1* Thus, every instance of a primitive solid type used to define an object, has associated with it a 4×4 transformation matrix which defines the appropriate scaling, rotation and translation operations to transform that instance of the primitive into the world coordinate system

The reason for using a local coordinate system in the first place is that the ray-primitive intersection calculations (*section 4 5*) can be greatly simplified by using the inverse of the object's transformation matrix to transform the ray from the world coordinate system back into the primitives local coordinate system The intersection test can then be conducted in the local system where the computational cost is greatly reduced, by virtue of the fact that we are then dealing with a unit sphere centered about the origin, a unit cube positioned along the positive XYZ axes *etc*, rather than an arbitrarily sized sphere or block centered about an arbitrary point in an arbitrary orientation This simplification results from the fact that the value calculated for t (the ray parameter) for the intersection of the transformed ray with the primitive, is the same as that for the intersection of the untransformed ray with the object, but is computationally less expensive to calculate

3.3.3 Ray Intersection And Classification

In the previous discussion of ray-object intersection in *section 2 2*, despite the fact that a ray may enter and leave a complex object at several successive points along the ray, only the closest point of intersection of a ray with the object was required This point could then be compared with the closest points of intersection for all other objects in the scene to determine the first object struck by the ray However, the situation with an object defined by a CSG representation is quite different since, the final solid is formed from addition, subtraction, or intersection of intermediate solids (called *composite* solids), which are in turn formed from similar operations on other composite solids Consequently, in order to know where a ray intersects a solid formed from say, the intersection of two other solids, A and B, we need to know where the ray enters *and* leaves solid A, and where it enters *and* leaves solid B For this reason, a ray is classified with a solid by a hst of t (ray parameter) values that correspond to the points on the ray at which it enters and leaves the solid

So, given a ray and a binary tree defining a solid composition, the ray is tested for intersection with the solid by recursively descending the composition tree (in postorder) from the top down to the leaf nodes, classifying the ray as in or out with respect to the primitive solids they represent, and then returning back up the tree, forming the classifications for the the composite (intermediate) solids by combining the classifications of left and right sub-trees The classification for the root then represents the in/out classification for the final solid with respect to the ray

3.3.4 Combining Classifications

Figure 3 5 illustrates how to combine left and right classifications for each of the intersection, union and difference operators (denoted by the symbols &, +, − respectively) by using solid lines to represent segments of a ray that are inside a solid and dashed lines for segments that are not As illustrated for the intersection operator in *fig 3 6*, the combination process is performed in three stages -

- The intersection points from the left and right rays are merged and sorted into ascending order to form a segmented composite ray
- Segments of this composite ray are classified as in or out in accordance with the combine operator and the classifications of the left and right rays (see *table 3 1*)
- Adjacent segments of the composite ray with the same classification are merged for simplification

3.3.5 Computational Cost

The algorithm as it stands, is something of a brute force method in that it tests all branches of the tree for intersection with the ray To appreciate the computational cost of such a scheme, consider an example using a solid composed of 100 primitive solids, displayed on a raster device of 500×500 resolution For such a resolution, firing one ray per pixel requires that 250,000 rays be generated, each of which must be tested for intersection and in/out classification with the solid Since the solid is composed of 100 primitive solids, its binary tree representation contains 100 leaf nodes, and therefore 99 internal nodes (say 100 for simplicity), giving

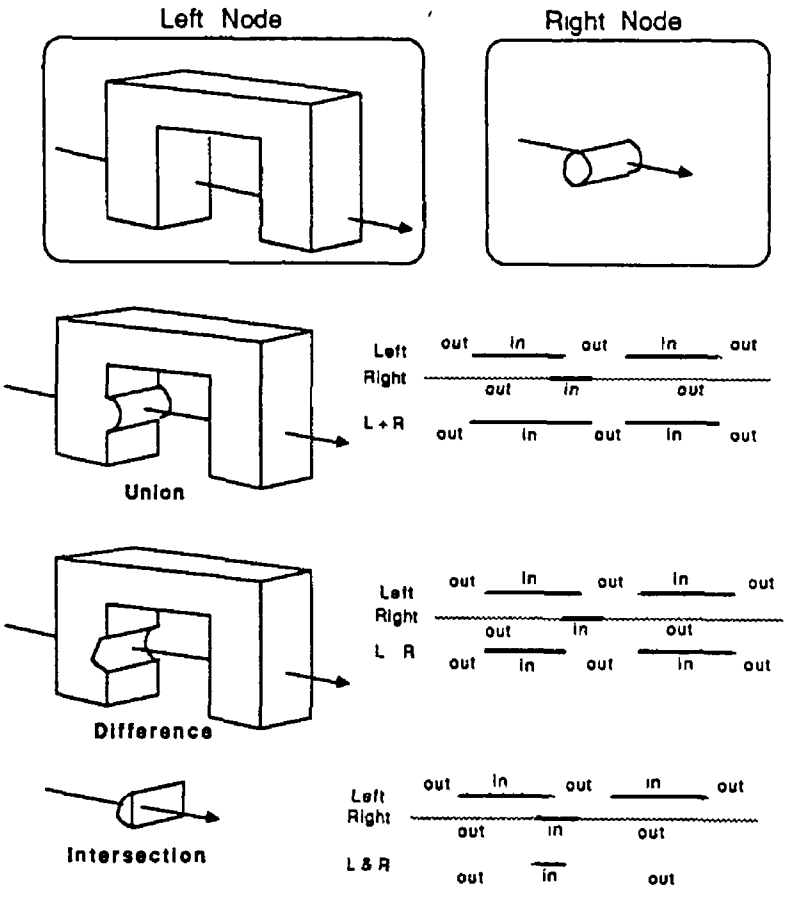
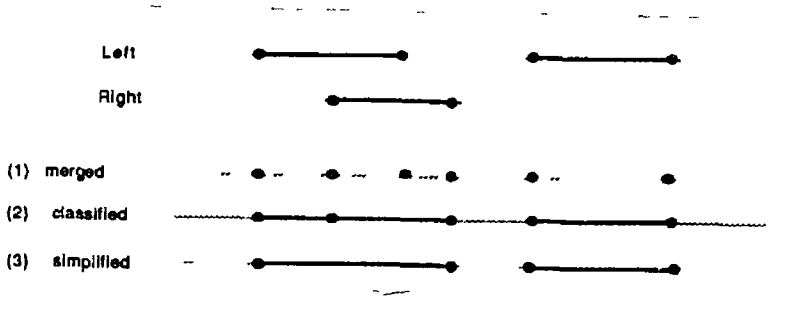


figure 3 5 Combining ray classifications



Three stage combine process for intersection

figure 3 6

OPERATOR	LEFT	RIGHT	COMPOSITE
intersection (&)	OUT OUT IN IN	OUT IN OUT IN	OUT OUT OUT IN
difference (-)	OUT OUT IN IN	OUT IN OUT IN	OUT OUT IN OUT
union (+)	OUT OUT IN IN	OUT IN OUT IN	OUT IN IN IN

Table 3 1 Ray Classifications

it a total of 200 nodes The four major areas of cost in rendering an image of such an object, namely the cost of recursive procedure calls, the cost of ray in/out classification, the cost of ray transformations, and the cost of ray primitive intersection testing, are itemised below -

- **200 x 250,000 = 50,000,000 recursive procedure calls**
  Each of the 250,000 rays visits each of the 200 nodes in the solid composition tree via recursion, requiring 50 *million* recursive procedure calls
- **100 x 250,000 = 25,000,000 ray in/out classifications**
  At each of the 100 internal nodes, classifications of the left and right branches must be performed, at a total cost of 25 *million* ray classifications
- **100 x 250,000 = 25,000,000 ray transformations**
  For each of the 100 primitive solids, the ray must be transformed into its local coordinate system, requiring a total of 25 *million* ray transformations

- **100 x 3 x 250,000 = 75,000,000 ray intersection tests**
  Testing a ray for intersection with each of the 100 primitive solids requires testing it with each surface of the primitive Allowing for an average of three surfaces per primitive solid (the sphere has 1, the cylinder 3, the block 6 *etc*), this represents a cost of 75 *million* ray surface intersection tests

TOTAL -	50 *million*	recursive procedure calls
	25 *million*	ray in/out classifications
	25 *million*	ray transformations
	75 *million*	ray-surface intersection tests

It it clear from these figures that there is a great need for some form of optimization The method chosen by Roth and described in the section below, is that of box enclosures

3.3.6 Box Enclosures — An Optimization

The use of box enclosures around the primitive and composite solids can provide a means by which non-contributing branches of the tree can be pruned from testing with the ray, thereby speeding up the intersection calculations

The scheme works by enclosing each primitive solid in a minimal bounding volume as it lies in its own local coordinate system (*section 2 4 1* discusses bounding volumes) Any transformations that are subsequently applied to an instance of the primitive solid are then also applied to its bounding volume so that, when transformed into the world coordinate system, the primitive is still enclosed by the transformed bounding volume

Box enclosures are then formed from these transformed bounding volumes by projecting them onto the screen and finding the minimum and maximum values of the projected X and Y coordinates (see *section 2 4 1, fig 2 12*) The minimum and maximum values of the unprojected Z coordinates are also determined for use in situations where rays may be bounded in depth A box enclosure is then defined by these two XYZ coordinate pairs, $(x_{min}, y_{min}, z_{min})$ and $(x_{max}, y_{max}, z_{max})$

Box enclosures are then calculated for the intermediate nodes in the tree by ascending the tree and combining the enclosures of the left and right sub-trees, in accordance with the boolean operator of the node *Figure 3 7* illustrates the calculation of the minimum and maximum X values of the combined box enclosure for each of the three boolean operators The minimum and maximum Y and Z values are calculated in a similar way Note that in the case of the intersection operator, the combined enclosure can be smaller than either of the enclosures from which it is formed

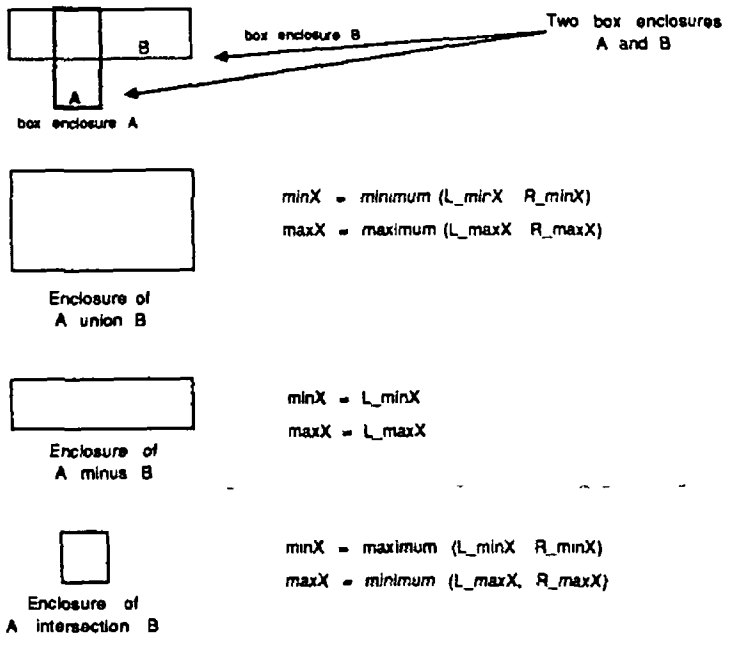


figure 3 7 Combining box enclosures

The reason for using screen projected box enclosures, rather than the original transformed bounding volumes, is that the the ray-enclosure intersection test is essentially a point in rectangle test, namely that the pixel spawning the ray lies within the rectangular screen enclosure of the projected bounding volume

By storing each box enclosure in its associated node, the solid composition tree not only contains a representation of the solid, but also a hierarchical representation of the space that the solid occupies So, at any node of the tree, if a ray fails to intersect the node's box enclosure, the sub-trees of that node can be eliminated from further processing Thus, the search for ray intersections resembles something of a binary search through the object space, rather than the exhaustive search required without the box enclosures

The savings provided by the use of enclosures is dependent both on the spatial distribution of the primitives and the organization of the solid composition tree, though according to Roth, the former is the more important of the two The ideal conditions for maximum effect would be that no primitive enclosures overlap in space and that the composition tree be balanced and organized in such a way that composite solids close to each other in space are also close to each other in the tree The worst situation on the other hand, would be where all primitive enclosures overlap each other

However, a situation where large numbers (or possibly all) of the box enclosures of the primitive solids mutually overlap is most likely to occur where a user is attempting to approximate a surface type not provided directly by the set of available primitive solid surfaces Such a situation can be more practically dealt with by adding a new primitive with the required surface type to the list of available ones

The use of box enclosures, while providing a good means of optimization, has the limitation that it only optimizes for primary rays This is because of the fact that the enclosures are specifically constructed in such a way as to take advantage of the fact that all primary rays originate at screen pixels Secondary rays however, such as shadow or reflected/refracted rays (*section 2 3*), originate at object intersection points, so their direction cannot be determined in advance For such cases, Roth suggests the use of spherical enclosures The main reasons for choosing spheres is that they are compactly defined by two values (a centre and radius) and have a relatively inexpensive ray intersection test — if the perpendicular distance from the ray to the centre of the sphere is less than the radius of the sphere, the ray intersects, otherwise it does not

3.3.7 Circumstance Classification

Even in cases where a ray does pass through the box enclosure of a node, it can still sometimes be possible to avoid unnecessary ray classifications under certain circumstances For example, if the operator at the node is either an intersection or a difference operator, and the ray classifies as being out of the left sub-solid (*i e* no intersection), the ray will classify as being out of that composite solid regardless of its classification with respect to the right sub-solid There is therefore no need to examine the right sub-solid In the case of the union operator however, the right sub-solid must still be processed

3.4 Further Optimizations

As outlined in *section 2.4*, the major part of the time taken to ray trace an image is taken up with testing rays for intersection with objects in the scene Optimization techniques therefore aim at trying to reduce the number of ray intersection tests that have to be carried out by employing hierarchical bounding volume, space subdivision and image coherence techniques Optimization techniques for objects defined by a CSG representation can be broadly placed into the same categories However, because of the nature of object definition in CSG, there are differences between the corresponding techniques for a CSG representation

For example, the box enclosure method outlined above could be described as a hierarchical bounding volume technique yet it differs from those outlined in *section 2.4.1* by virtue of the fact that the hierarchical structure is embedded into the object definition structure itself, namely the object composition tree

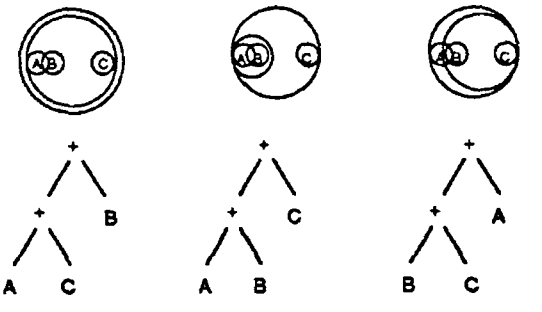
3.4.1 Enclosures And Tree Rearrangement

The box enclosures implemented by Roth and outlined above are a specialized form of bounding volume, and have the limitation that they can only be used for primary rays Gervautz [GERV86], uses spherical enclosures as advocated by Roth, as well as rectangular enclosures whose planes lie parallel to the XZ, YZ and XY planes These rectangular enclosures differ from Roths box enclosures in that the latter are calculated by projecting the transformed primitive bounding volumes onto the screen *before* taking minimum and maximum X and Y coordinate values, whereas the former are calculated by taking minimum and maximum values *without* first projecting As a result, the Gervautz enclosures can be applied to both primary and secondary rays, but have a more costly intersection test than Roths Like Roth however, Gervautz forms enclosures for intermediate nodes by combining enclosures for the left and right sub-trees in accordance with the nodes boolean operator

Gervautz also employs the commutative and distributive properties of the union operator to rearrange sub-trees containing only union operations The reasons for rearranging the tree in such manner are twofold -

- The tree can be made more symmetrical, resulting in fewer recursive calls to the intersection procedure

- Rearrangement can result in smaller enclosures in the internal nodes, resulting in fewer unnecessary ray intersection tests with the sub-trees of those nodes, *fig 3 8*



Three possible tree arrangements for the three objects A, B and C The centre one is best since it results in the smallest enclosure for the left subtree

figure 3 8

3.4.2 Scan-Line Enclosures And Active Trees

From *figure 3 7*, it can be seen that the union operator is the only one of the three operators which results in a box enclosure that is larger than either of the left or right enclosures from which it is composed — for the intersection operator it is smaller than either left or right enclosures while for the difference operator it is the same size as the left enclosure In addition, from *fig 3 9* it can be seen that an object formed from the union of two sub-objects can often have a box enclosure that contains large volumes of empty space The damaging effect of such a situation on the efficiency of enclosures can be even greater if it occurs near the leaf nodes of the solid composition tree, since it can then have a cumulative effect on enclosures at other union nodes higher up in the tree

SCAN-LINE ENCLOSURES Broonsvoort [BRON84], overcomes this problem of box enclosures by using interval enclosures instead Interval enclosures are very similar to box enclosures, except that the latter refer to rectangular areas on the screen while the former refer to intervals along the current scan line of pixels So, instead of storing minimum and maximum X and Y values at each node, defining a rectangular area on the screen enclosing the primitive or composite solid represented by the node, only a minimum and maximum X value is stored,

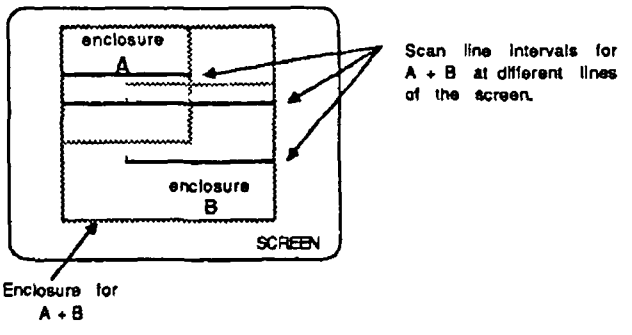


figure 3 9 Box and scan- line interval enclosures

corresponding to the section of the current scan line in which the primitive or composite solid represented by the node is contained, *fig 3 9* The generation of the interval enclosures is also similar to the that of box enclosures in that they are first explicitly calculated for the primitive solids and then calculated for composite solids by applying the corresponding operator to the interval enclosures of the left and right sub-trees

Unlike box enclosures however, which remain constant throughout the entire rendering process, interval enclosures for composite solids can vary from one scan line to another and so have to be recalculated To avoid unnecessary recalculation, box enclosures are initially computed for the primitive solids The minimum and maximum X values of each box are then collected and stored in the corresponding leaf nodes and the interval enclosures for composite nodes calculated as above Then the minimum and maximum Y values of each box are collected and sorted into ascending order in an array At the transition from one scan line to the next, the array is checked to see if a maximum or minimum Y value has been crossed If one hasn't, then the interval enclosures are the same for the new scan line as for the previous one, otherwise they have to be recalculated

By recalculating interval enclosures at the appropriate times, Broonsvoort manages to bypass non-contributing nodes of the CSG tree during each traversal For this purpose an *active* CSG tree, which omits these non-contributing nodes, is generated each time interval enclosures are recalculated This active CSG tree is implemented through the use of additional pointers in each composite node of the solid composition tree, that point to the left and right active sub-trees

3.4.3 Bounding Ray Depth

A different optimization implemented by Gervautz [GERV86] which can be used independently of the enclosures outlined previously in *section 3.4.1*, is that of only testing for intersection along certain sections of a ray, called a *bounded ray*. As outlined in *section 2.2*, a ray is conveniently expressed by a line equation in parametric form, where points on the line/ray are ordered and accessed via a parameter t, which can mathematically take on values of minus infinity to infinity. However, in some circumstances (outlined below), it makes sense to exclude certain sections of this infinite ray from intersection testing by limiting the value of t.

User Limit In some situations, a user may only want an image of objects that are near to the viewer, *i.e.* objects that are far to the back of the scene are to be omitted from the image. This can be achieved by the user specifying the distance behind which objects are to be excluded. This distance value can then be used to limit/bound the ray in depth so that objects in the excluded zone are not tested for intersection. Even in cases where no bound is specified, since it is unlikely that the user wants to see objects behind the view-point, the ray can still be bounded by limiting t to positive values only.

Shadow Rays Testing if a point on a surface lies in shadow with respect to a light source involves casting a ray from the point to the light source. Since objects beyond the light source with respect to the direction of the ray cannot cast a shadow on the surface, the ray can be bounded to the section from the point on the surface to the light source.

Dynamical Bounding For intersection and difference operators, only those parts of the right sub-tree that overlap the left sub-tree are of interest. Consequently, only that section of the ray which intersects the left sub-tree need be tested for intersection with the right sub-tree.

3.4.4 Temporary Object Trees

As seen earlier, the use of box enclosures embedded into the nodes of the solid composition tree can reduce the computational cost of rendering an image of the solid by providing a means by which non-contributing sub-trees can be bypassed in the search for ray-solid intersections. A sub-tree can only be bypassed however, if the ray misses the box enclosure of the sub-tree, which can only be determined by testing the ray for intersection with the box enclosure.

If on the other hand, there were a means of knowing in advance that particular sub-trees would not be intersected by the ray, a new temporary solid composition tree could be generated for the ray, which excluded any non-contributing sub-trees The ray could be intersected with this temporary tree more rapidly and efficiently by virtue of the fact that it would have fewer nodes and possibly shallower depth, resulting in fewer recursive calls and intersection tests Unfortunately, the cost of generating such a temporary tree for each ray would probably far outweigh any computational savings it provided However, if large groups of rays could be found that shared a common temporary tree, then savings would be possible The temporary tree for a particular ray could then be found by determining the group to which the ray belongs and using its associated temporary tree

Gervautz [GERV86] outlines a method for generating such temporary trees and the groups of rays that share a common one To a certain degree, the method resembles something of a 2D space subdivision in screen space, in that the screen is partitioned into a number of rectangular regions, each of which has associated with it a temporary tree containing only those primitive and composite solids whose box enclosures project onto the specified region A quadtree structure is used to administer the temporary trees, with the image being rendered rectangle by rectangle, by traversing the quadtree and rendering all pixels associated with the rectangular region represented by each node Rectangles that do not contain any primitive or composite solids will have an empty tree associated with them and thus can be trivially processed by setting all of their pixels to the background colour

The rectangles are generated by first projecting a primitive solid enclosure (*section 3 4 1*) onto the image plane The minimum and maximum X and Y values of the projected enclosure then define four halfplanes, $x > x_{max}$, $x < x_{min}$, $y > y_{max}$, $y < y_{min}$, and the projection of all primitive solid enclosures in such a fashion produces an irregular rectangular grid pattern on the screen For each rectangular grid region, a temporary composition tree is formed from the original one, containing only those primitive solids whose enclosure projections mapped partly or completely into the region, *fig 3 10*

Using the rules of *table 3 2*, it is possible to eliminate both non-contributing internal and primitive nodes from the original solid composition tree (E represents the node to be eliminated, A represents the other node) -

Because of the fact that the primitive solid enclosures were projected onto the viewing plane, the rectangles and temporary trees discussed above can only be applied to primary rays However, by projecting them onto a plane that

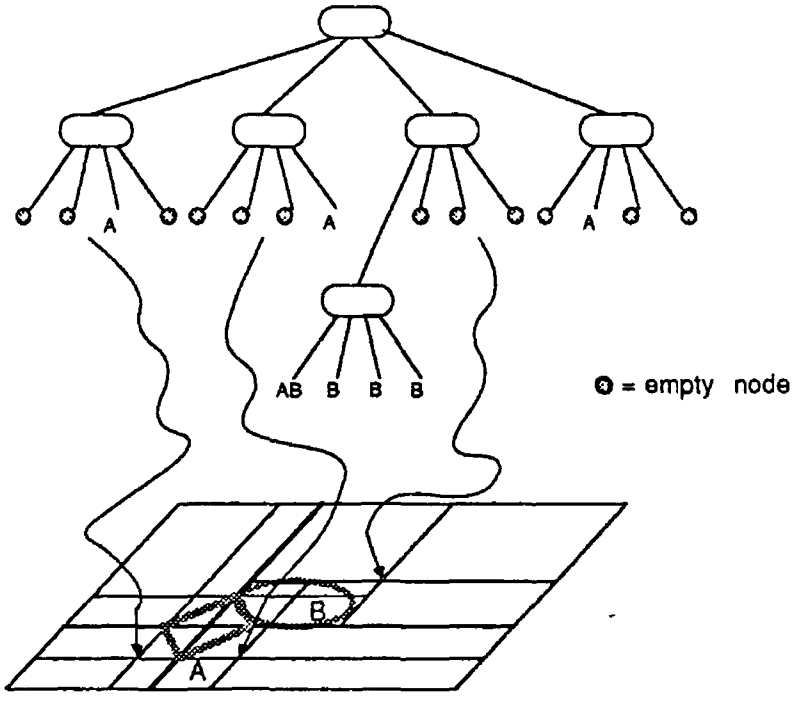


figure 3 10 Quadtree administration of tempory object trees

lies between the light source and the scene, a similar quadtree structure can be generated which can be used to optimize the tracing of secondary rays to the light source Tracing for example, a shadow ray to the light source would simply involve determining the rectangle through which the ray passes and intersecting it with the associated temporary tree of that rectangle

The case for reflected and refracted rays however is a little more complex than that for primary and shadow rays While all primary rays are constrained to pass through the viewing point, and all shadow rays are constrained to pass through the light source, there is no single point through which reflected and refracted rays are constrained to pass Consequently, there is no single 2D rectangular grid that can be generated which is relevant to all reflected/refracted rays in the same sense that there is for primary and shadow rays Therefore, to optimize for such rays, a 3D grid of cells administered through an octree structure has to be generated in place of a 2D grid of rectangles and quadtree structure The three dimensional cells and their associated temporary trees are generated using the planes of the primitive solid enclosures (which, as outlined in *section 3 4 1*, lie parallel to the XZ, YZ and XY planes) in similar fashion to the way in which the 2D projections of the enclosures are used to generate the 2D grid and temporary trees

Operation	Action
UNION (A + E)	eliminate node A
DIFFERENCE (A - E)	eliminate node A
DIFFERENCE (E - A)	eliminate node A and minus node
INTERSECTION A & E)	eliminate internal intersection node

Table 3 2 Eliminating of non-contributing nodes

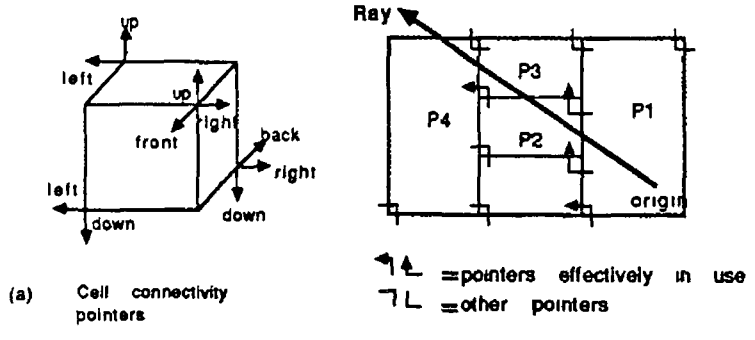
3.4.5 Space Subdivision

Arnaldi [ARNA87] outlines a method of dividing space into 3D cells, into which a solid composition tree can be distributed such that each cell contains a minimal CSG tree consisting only of those primitives from the original tree that are relevant to the cell Intersecting the ray with the original solid then involves propagating the ray from one cell to the next (along the ray direction), and testing it with the cells associated CSG tree Since the 3D space is divided in such a way that the cells fit as closely as possible the primitive solids, the associated composition tree for each cell should be significantly smaller and hence, faster to intersect that the original tree

In order to provide fast and efficient ray propagation from one cell to the next, each cell, in addition to its minimal composition tree, has an associated set of connectivity information that takes the form of a list of pointers to neighboring cells Initially, each face of every cell had an associated list of pointers containing a pointer to every cell adjacent to the face The drawbacks associated with this method however were firstly, given the non-uniformity of the cell sizes, some faces of some cells required longer lists that others, which meant that face lists had to be dynamically allocated in size Other problems were that in order to ensure complete connectivity, many cells contained redundant pointers and that

propagation of a ray through a face of a cell involved an inefficient linear search through its associated pointer list

Arnaldi overcame these difficulties by adopting a *corner stitching* technique used in the design of 2D VLSI layouts and extending it to three dimensions Using this method, each cell has a fixed number of pointers (10) associated with it that connect cells together through their corners, *fig 3 11a* Thus, while all cells adjacent to a given cell face are not directly connected by these pointers, it is still possible to get to any one of them by taking an indirect "corner route" through the appropriate cells, *fig 3 11b*



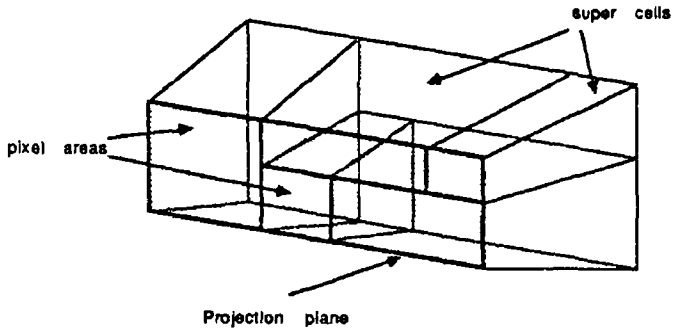
(a) Cell connectivity pointers

(b) 2D projection of cell structure showing a ray passing successively through cells P1 P2, P3 and P4

figure 3 11

The cells themselves are formed by a two stage process, 2D subdivision of a projection plane, followed by its extension to 3D, *fig 3 12* The 2D subdivision is similar to that performed by Gervautz [GERV86] outlined in the previous section Firstly, a minimal bounding volume associated with each primitive is projected onto the viewing plane, producing a set of rectangles on the plane Each rectangle is then decomposed into four segments which are used as the basis for a binary space partitioning process that results in a 2D partitioning of the viewing plane, whose extension along the Z-axis gives rise to 3D cells, called *supercells* Each supercell is then subdivided into smaller cells along the front and back planes of any primitive solid bounding volumes enclosed in the cell

MAILBOX As outlined in *section 2 4 2*, the general idea behind space subdivision schemes is to divide space up into cells and associate with each cell a list of all objects either completely or partly enclosed in the cell As the ray passes from one cell to the next, it need only be tested for intersection with the list of objects associated with the cell Consider however, the case where an object is partly enclosed by several different cells Each of these cells will then have



Creation of 3D cell structure by extending 2D screen partitions to three dimensions

figure 3 12

the object in their associated list Since it is quite possible that a ray may pass through several of these cells, the object will be tested for intersection more than once with the same ray, which is both unnecessary and wasteful

Arnaldi overcomes this problem, which is common to all space subdivision schemes, by associating a "mailbox" with each primitive and a unique number with each ray The mailbox structure then stores the intersection point (if any) and ray number of the last ray that was tested for intersection with the primitive Then, before testing the primitive with a particular ray, a test is made to see if the ray number in the mailbox is the same as the one for the current ray If it is, this primitive has already been tested with this ray and the result can be read directly from the mailbox If the number is different, the intersection test goes ahead and the ray number and intersection information in the mailbox are updated

Chapter 4

MicroTrace

Ray tracing, despite its elegant approach to image synthesis and its realistic image generation capabilities, has tended to be confined to the realms of large mainframe computers by virtue of its large appetite for computation This chapter however, discusses in detail an implementation of a ray tracer called *Micro Trace* which has been designed, written and developed on a microcomputer A presentation of its results and an evaluation of its performance is presented in *section 4 8*

Before taking a detailed look at the ray tracer in *section 4 2*, the following section gives a brief description of the microcomputer and display device on which the research was carried out However, despite having been implemented on a specific type of microcomputer with a specific type of display device, the ray tracer has been designed with portability in mind Machine independence has been facilitated by writing source code that complies with the ANSI C standard, while display device independence has been enhanced by providing an option of tracing an image to a file, in a format that can be customized for any sort of raster device

4.1 Hardware

The microcomputer consists of an IBM AT with a *Professional Graphics Display* and the following specifications -

- Intel 80286 processor running at 6 *MHz*
- Intel 80287 maths coprocessor
- 640K of main memory
- 20 *Mb* disk storage

4.1.1 Professional Graphics Adapter

The Professional Graphics Display is a high quality colour raster display with a resolution of 640×480 pixels (horizontal x vertical) It uses four bits per primary for displaying colours, giving it a capability of distinguishing 4096 different colours (*section 1 3 2*) The display is controlled by a Professional Graphics Adapter (PGA), a VLSI card which contains the refresh buffer and a host of built in graphics functions, to which end it contains its own 8-*bit* processor, an Intel 8088 From the schematic layout of the adapter in *fig 4 1*, it can be seen that the microcomputer communicates with the adapter via an input/output buffer It can also be seen from the diagram that the display refresh buffer is contained on the adapter card and does not form part of the AT's address space Consequently, the AT cannot directly address this memory and instead must read and set pixel values by issuing the appropriate commands to the adapter via the input buffer, reading results back from the output buffer

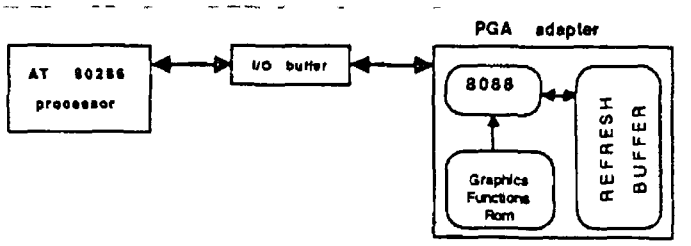


figure 4 1 Schematic layout of IBM AT and PGA

As mentioned earlier, the PGA contains a number of built in graphics functions for drawing lines, circles polygons etc which again are executed by sending the appropriate command and parameters (in a predefined hexadecimal format) to the input buffer In addition to these 2D commands, the adapter also has a

number of 3D graphics commands that automatically apply clipping and projection to 3D lines and polygons In implementing the ray tracer however, this 3D capability of the PGA has been deliberately avoided and only those capabilities usually available for most other display devices, such as reading/writing pixel values and 2D line drawing, have been used

LOOK-UP TABLE As mentioned above, the Professional Graphics Display has a resolution of 640×480 pixels with 4 *bits/primary* (or 12 *bits/pixel*) Instead of storing each pixel explicitly as 12-*bits*, which would require a refresh buffer of 450K (640 × 480 × 12 *bits*) the adapter stores each pixel as a single byte which is then used as an offset into a 256 element look-up table containing the 12-*bit* RGB value for the pixel, *fig 4 2* While this arrangement reduces the refresh buffer requirement to 300K, it means that only 256 of the 4096 possible colours can be displayed on screen at any one time The user can select any 256 of the 4096 colours by appropriately loading values into the look-up table via commands sent to the adapter input buffer Colours of lines, polygons *etc* are then specified as values in the range 0 255 with the actual colour being determined from the value in the look-up table

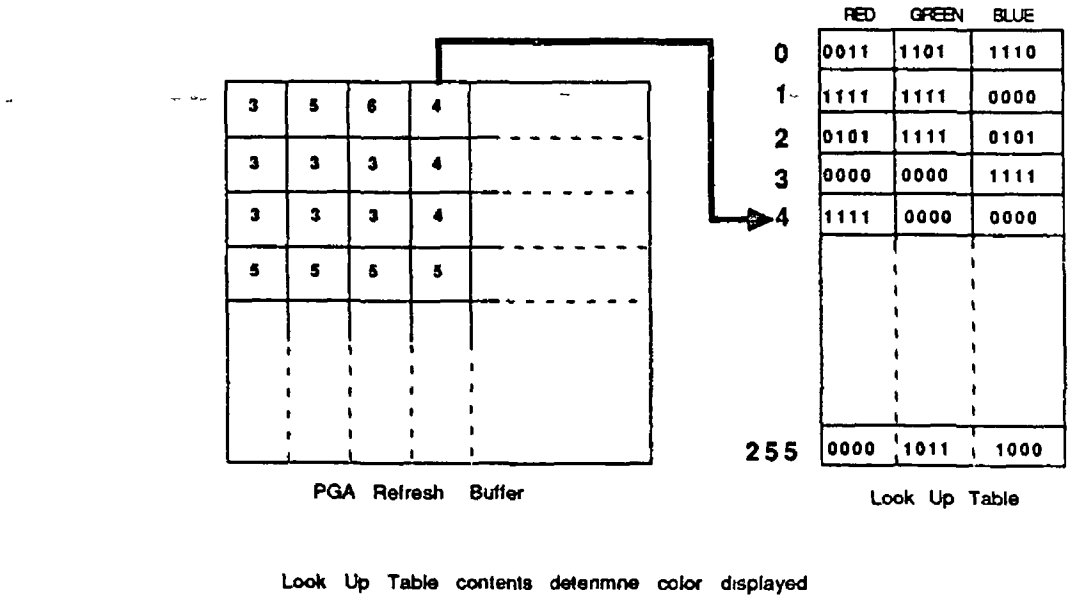


figure 4 2

PGA LIBRARY As a preliminary step in building the ray tracer, a *C* function was written for each PGA command, which packages the appropriate parameters in the correct predefined PGA hexadecimal format, places them in the adapter input buffer along with the appropriate command code, and reports results/errors

placed by the adapter in the output buffer These functions were then placed in a program library, allowing PGA commands to be easily incorporated into any *C* program simply by calling the corresponding *C* function

4.2 *Micro*Trace — The Inner Workings

While, as mentioned earlier, a reasonable degree of machine and display device independence were considerations in the design of *Micro*Trace, as was easy incorporation of certain future enhancements to the ray tracer (*section 5 2*, the primary aim was to examine the practicality and feasibility of microcomputer ray tracing, a question which is addressed in the following chapter The concern of this chapter is to describe in detail the various functional modules and the overall design and implementation of *Micro*Trace Although the ray tracer has been written in *C* (*Microsoft C* version *5 1*), a knowledge of the language is not a prerequisite for an understanding of the following sections

4.2.1 A Brief Overview

*Micro*Trace is essentially a primitive instance rendering system (*section 3 1*) schematically represented in *fig 4 3*, whose set of primitives consists of a cube, a sphere, a cylinder and a cone A "scene" to be rendered is presented to the ray tracer as a linked list of objects, built up by the user through the application of scaling, translation and rotation operations on the primitive types These operations are performed by functions provided in the user interface module, which also provides various other sets of functions for defining scene environment elements such as parallel/perspective view, light source position and intensity, view plane *etc* and for defining image output/format, optimization options *etc*

An image of the scene can be generated in either a perspective or parallel view (*section 1 5 2*) on any viewing plane perpendicular to the *Z*-axis Shadows can also be incorporated into the image At present, only a single light source can be specified, but it can be specified to be either a directional light source (all light rays are parallel to the specified direction) or a point light source positioned at a specific *XYZ* coordinate in the scene The current lighting model (Phong's model, *section 1 7 2*), does not incorporate transparent objects but does account for ambient, diffuse and specular lighting components

The ray tracer functions in one of two modes, either PGA or RGB mode

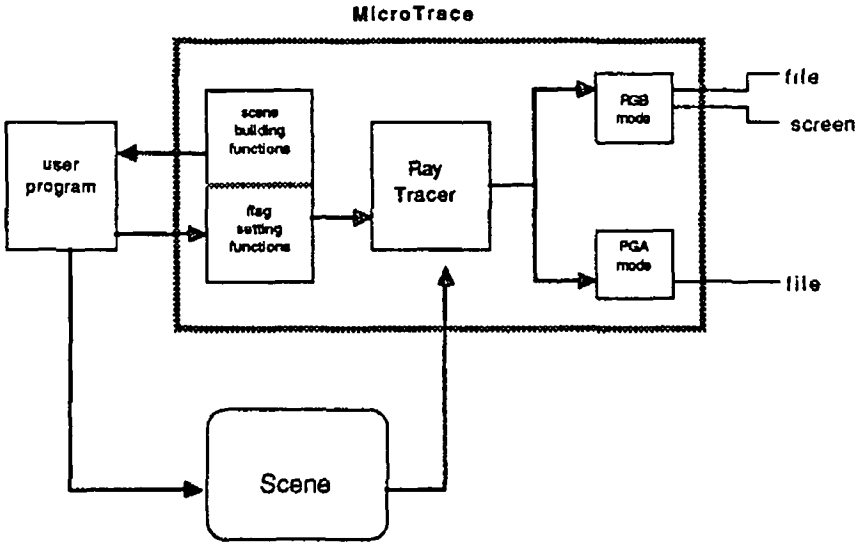


figure 4 3 Schematic diagram of MicroTrace

The PGA or Professional Graphics Adapter mode allows both file and screen image output options and generates images in a format suitable for display on a PGA display The RGB or Red Green Blue mode on the other hand has only a file output option and produces images in a file format that can be customized for display on any type of RGB display

4.2.2 PGA Mode

This is the default mode for *MicroTrace* and allows the option of displaying the image on a Professional Graphics Display (if one is available) as it is being generated and/or of writing the image to a file in a format that can later be directly displayed on one The default settings of this mode assume that a PGA card and display is attached and generate only a screen image However, these settings can easily be changed, using functions in the user interface module, to any combination of file and display options That is to say, it is possible to have

[1] screen but no file output (default)
[2] file but no screen output
[3] screen and file output
[4] no output at all

Option [2] provides for the generation of PGA images on a machine which does not have a PGA card and display attached to it, while option [4] is used to avoid discrepancies that could result from differences in disk/display speeds on different machines when comparing image generation times

In PGA mode two output file formats are available, compressed format and uncompressed format (compressed is the default) In compressed format the image is compressed using a PGA run length encoding method that can substantially reduce the size of the image file, while in uncompressed format the image is essentially stored as one byte per pixel

Uncompressed File If the uncompressed format is selected, the image file generated will not be run length encoded but will have the following format -

FF	x1 x2 y1 y2	DATA

The first byte of the output file, hex **FF** is simply an identification byte to indicate that the file is in uncompressed form There then follows 8 bytes which represent the x_{min}, x_{max}, y_{min} and y_{max} screen coordinates respectively of the viewport specified for the image Each of the four coordinates is stored as a two byte integer in backword format (least significant byte first[1]) Then follows the data in the form of one byte per pixel in viewport left to right, top to bottom order, which represents the PGA colour of the pixel (*section 4 2 3* outlines how these intensities are calculated)

As outlined in *section 4 1 2*, when the file image is displayed, the actual colour for each pixel will be determined from the corresponding 12-*bit* RGB entry of the 256 element look up table This table is stored in a separate file consisting of 256 two byte integers (stored least significant byte first) which contain the RGB values in the form shown below (*bits* 12,13,14 and 15 are "don't cares") -

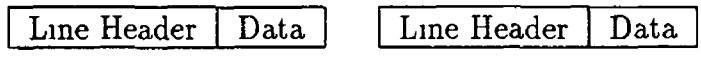
COLOUR	BITS
red	11 10 9 8
green	7 6 5 4
blue	3 2 1 0

While it may be possible to display the image contained in such a file on a display other than a PGA display, by using the look up table in conjunction with

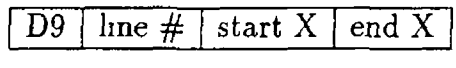
[1] If the machine on which the image is generated does not use back word storage format, they will be stored normally *i e* most significant byte first

the image file to access and modify pixel RGB values, a more readily adaptable format can be produced by setting *Micro Trace* into RGB mode where RGB values are explicitly stored for each pixel in the image file

Compressed File If the compressed file format is specified (default) in PGA mode, a run length encoding system is used to reduce the size of the image file resulting in the following file format -

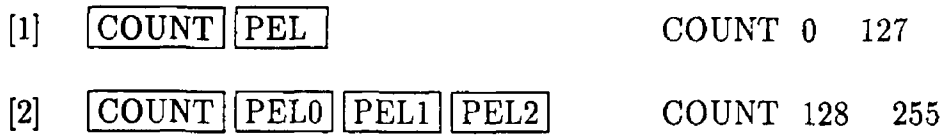


The line header which precedes each set of data, has the form -



The **D9** is the hexadecimal PGA code command code for "write encoded line of pixels" Then follows three integer numbers, each stored as two bytes (least significant byte first), representing the vertical line number and the horizontal screen coordinates where this line begins and ends Note that there is no explicit *id* byte or viewport at the beginning of the file since, in this format the first byte will always be D9 and the viewport is encoded as part of the image data for each line

The data for each line consists of a series of packets, of which there are two kinds -



If the count is in the range 0 127 then the byte that follows is the colour of the next COUNT+1 pixels Alternatively, if the count is in the range 128 255 then the next COUNT-127 bytes that follow are the colours of the next COUNT-127 pixels As in uncompressed format, these pixel values represent PGA colours in the range 0 255 (*section 4 2 4* outlines how these intensities are calculated) The actual colours are determined, as before, from the look up table entries, which are stored in a separate file

Compressed *Vs* Uncompressed In most cases, compressed form is preferable to uncompressed form for two reasons Firstly, an image file in uncompressed format is quite likely to be substantially larger than its corresponding compressed counterpart Secondly, displaying an image from an uncompressed file takes longer by virtue of the fact that the PGA only accepts a line of pixel values in compressed format Hence, each line of pixels read from the uncompressed file has to be run length encoded by *MicroTrace* before being sent to the PGA input buffer (*section 4 1 2*) whereas the compressed file can be simply read and sent directly there

However, uncompressed format can sometimes be useful in modifying a file image without first displaying it, since the exact offset of a single pixel value in the file can be easily calculated, whereas in compressed format, finding a particular pixel involves reading and interpreting the run length encoding

4.2.3 RGB Mode

RGB mode provides only a file output option (no output can also be specified) The no output option, as for the PGA mode, is used to avoid discrepancies that could result from differences in disk speeds on different machines when comparing image generation times

When a file output is specified, the output file format is the same as that for the uncompressed PGA format described above, except that the identification byte is hex **00**, and the data consists of three bytes per pixel instead of one The three bytes represent the red green and blue intensities for the pixel, each of which is a value in the range 0 255, giving a colour palate of 16 *million* colours These red green and blue intensities, as outlined in the following section, are calculated in quite a different way to pixel intensities calculated in PGA mode

4.2.4 Calculating Pixel Intensities

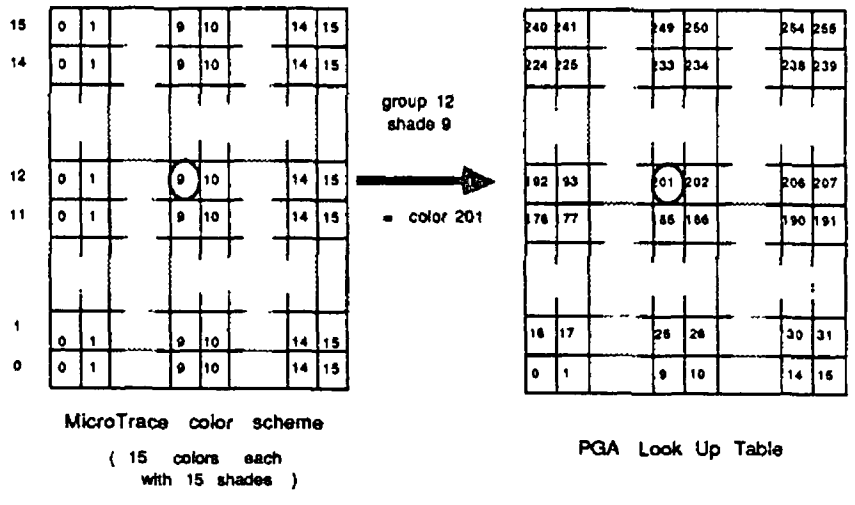
Sections 4 2 2 and *4 2 3* above have outlined the two different operating modes of *MicroTrace* Both modes are essentially the same in the respect that in either mode, the same ray tracing functions are used to determine the closest object of intersection for each pixel and the same lighting model is used to determine pixel colour and intensity However, even though the same lighting equation is applied in both modes, the way in which it is applied and interpreted in the two modes differs significantly

Before looking at how the equation is applied in PGA and RGB modes however, the equation is first summarized below for reference A detailed explanation of the terms and constants of the equation can be found in *section 1 7 2*, which discusses the Phong lighting model in detail The intensity of light I, emitted from a point on a surface with ambient, diffuse and specular constants of K_a, K_d and K_s respectively, is -

$$I = I_a K_a + I_d K_d + I_s K_s$$

where I_a is the constant ambient light intensity and I_d and I_s are the calculated diffuse and specular intensities at the point

PGA MODE In PGA mode, the look up table is loaded with 16 different shades of 16 different colours, *fig 4 4* The colour of an object, which is contained in the *clr* field of the structure for the object (*table 4 1*), is then specified as an integer value in the range 0 15, corresponding to one of the 16 colour groups in the look up table Calculating the colour of the light emitted from a particular point on a surface involves applying the above lighting equation to generate an intensity value in the range 0 1 for the point This value is then converted into an integer value in the range 0 15 and used as an offset into the colour group of the object, resulting in a colour in the range 0 255



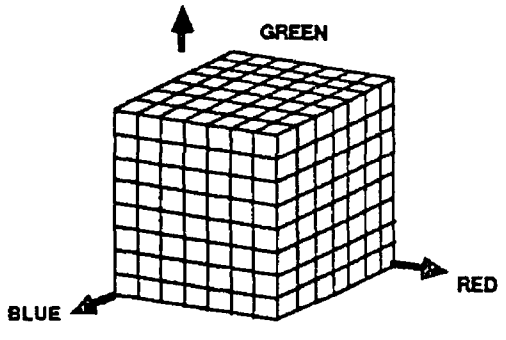
In PGA mode objects are shaded using the 16 shades of whichever of the 16 colors they have been assigned

figure 4 4

Example see *fig 4 4*

object colour	⟶	12	
calculated intensity	⟶	0 6	Ia Ka + Id Kd + Is Ks
converted intensity	⟶	9	15 x 0 6
final colour	⟶	201	16 x 12 + 9

The shades for a given colour group are loaded by the user before ray tracing commences, through one of the functions provided in the user interface module For each of the 16 colour groups, the user specifies to the function a single RGB value, from which the function generates 16 shades for the colour group This is achieved by interpreting the supplied RGB value as the XYZ location of a cell in an imaginary cube consisting of 4096 ($16 \times 16 \times 16$) such cells, *fig 4 5*, and following a line from the origin through this cell (and beyond if necessary) until it has passed through 16 cells of the cube The XYZ coordinates of these 16 cells form the RGB values of the 16 shades for the specified colour group Alternatively, a different function can be used which, instead of following a line from the origin upwards, follows it from the diagonally opposite corner, downwards Thus, the shades for a particular colour group can be forced to include either white as the highest intensity or black as the lowest (only a colour group specified by a cell of equal X, Y and Z coordinates contains both)



Each of the 4096 possible PGA colors can be viewed as a subcell of a 16x16x16 cube rested on red, green and blue axes

figure 4 5

RGB MODE In RGB mode, the red, green and blue components of the light emitted from a particular point on an object are calculated by first calculating it's cyan, magenta and yellow (CMY) components and converting them to RGB The reason for using the subtractive primary colours, CMY, is that reflection of light is essentially a subtractive process So, in the same way that the RGB

intensities for any colour can be interpreted as the respective intensities of red, green and blue light which, when added to black give the specified colour, the CMY intensities of the colour are the respective intensities of cyan, magenta and yellow, which, when subtracted from white light, give the same colour

These CMY intensities are calculated for a point on a surface by applying the lighting equation three times, once for each of cyan, magenta and yellow, to give an intensity for each in the range 0 1 These intensities are then converted to RGB intensities, each one an integer in the range 0 255

Example

$$\begin{aligned} \text{red} &= 255 \times (1 - I_a K_a + I_d K_c + I_s K_s) \\ \text{green} &= 255 \times (1 - I_a K_a + I_d K_m + I_s K_s) \\ \text{blue} &= 255 \times (1 - I_a K_a + I_d K_y + I_s K_s) \end{aligned}$$

4.2.5 The Object Structure

Micro Trace implements a set of four basic primitive types, *fig 4 6*, each one defined in its own local coordinate system as follows -

CYLINDER	cylinder of unit radius and length centered horizontally along the Z-axis
SPHERE	sphere of unit radius, centered about the origin
CUBE	unit cube standing on positive XYZ axes
CONE	cone of unit height with base of unit radius, centered horizontally along the Z-axis, with its apex at the origin

An object, which is an instance of one of these basic primitive types, consists simply of a basic primitive name, and a 4×4 transformation matrix which describes the translation, rotation and scaling operations that transform the primitive from its own local coordinate system into the world coordinate system

The 4×4 transformation matrix, along with the basic primitive type name

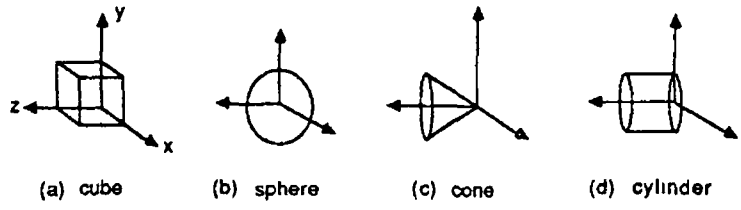


figure 4 6 The four primitive solids of MicroTrace

are stored in a *C* structure, with the user interface module providing functions for allocating space for such an object structure, and for building up the transformation matrix A scene consisting of any number of such objects is then presented to be ray traced as a linked list of these structures

In addition to storing a primitive type name and transformation matrix, the structure has several other fields, some of which are filled in by the ray tracer These fields contain various other items of information about the object required by the ray tracer and the shader *e g* its colour A complete list and description of the fields of the object structure is given in *table 4 1* below, though the function of some of them may not be apparent until the areas of ray-object intersection and optimizations have been covered in *sections 4 5* and *4 6* respectively

4.3 Ray Generation

As mentioned in *section 2 2*, a ray can be conveniently represented as a line in parametric form, defined by a point $P(x_0, y_0, z_0)$, a direction vector $D(D_x, D_y, D_z)$ and a parameter t All points on the line are then ordered and accessed via t with each point (x, y, z) on the line given by -

$$\begin{aligned} X &= X_0 + tD_x \\ Y &= Y_0 + tD_y \\ Z &= Z_0 + tD_z \end{aligned} \tag{4 1}$$

Positive increasing values of t give points on the line that are increasingly further along the line from (X_0, Y_0, Z_0) in the direction (D_x, D_y, D_z), while negative

Field Name	Description
primitive	name of the primitive type from which the object is derived
transform	4 × 4 transformation matrix
inverse	4 × 4 inverse transformation matrix
xmin xmax ymin ymax	screen rectangle (*extent*) enclosing the object's projected bounding volume
znear zfar	nearest & furthest Z coordinates of object's transformed B V from primary ray origin
clr	object's PGA colour group (0 15)
rnd	roughness of the object's surface (used as a coarse approximation of a textured surface)
ka kd ks	object's ambient, diffuse and specular reflection constants
pwr	object's specular power constant
cmy	object's cyan, magenta and yellow reflection ratios
next	pointer to next object in list

Table 4 1 The Object Structure

decreasing values give points that are further in the opposite direction For this reason, the point (X_0, Y_0, Z_0) is often referred to as the ray origin

An image is then ray traced by casting a ray through each pixel in the screen, determining the closest object struck by the ray and calculating the shade of the object at the point of intersection, which then becomes the colour for the pixel

In *Micro*Trace, the ray equation for a given pixel is determined from the perspective/parallel viewing parameters, the viewport, the window and the view plane The first stage in determining the ray equation for a pixel is to map the pixel onto the view plane, where the view plane can be any plane parallel to the XY plane, *i e* $Z = dist$, (where $dist$ is the distance of the plane from the origin) So, given the following viewplane, window, and viewport -

viewplane	$\longrightarrow$	Z=dist
viewport	$\longrightarrow$	v_{x1} v_{x2} v_{y1} v_{y2}
window	$\longrightarrow$	w_{x1} w_{x2} w_{y1} w_{y2}

a pixel (X_v, Y_v) on the screen maps onto the point $(X_w, Y_w, dist)$ on the window where -

$$X_w = w_{x1} + (X_v - v_{x1})\frac{w_{x2} - w_{x1}}{v_{x2} - v_{x1}}$$

$$Y_w = w_{y1} + (Y_v - v_{y1})\frac{w_{y2} - w_{y1}}{v_{y2} - v_{y1}} \qquad (4\ 2)$$

This mapped point $(X_w, Y_w, dist)$ forms one half of the ray equation, the ray origin The second part, the ray direction is determined from the parallel/perspective viewing parameters If the view is defined to be a parallel projection type view *i e* the viewer is positioned at infinity along a specified direction then all primary rays are parallel to the given direction (P_x, P_y, P_z) On the other hand, if it is a perspective projection type view then the viewer is at a given point (X_v, Y_v, Z_v) and all primary rays will have a slightly different direction, see *section 2 2* The ray direction (D_x, D_y, D_z) can then be determined for a perspective or parallel view by -

$$\text{perspective} \quad (D_x, D_y, D_z) = (X_w - X_v, Y_w - Y_v, dist - Z_v)$$

$$\text{parallel} \quad (D_x, D_y, D_z) = (P_x, P_y, P_z)$$

This ray equation, defined by the point $(X_w, Y_w, dist)$ and the direction (D_x, D_y, D_z), is then tested for intersection with the list of objects in order to determine the closest object (if any) struck by the ray

4.4 Transforming The Ray

In testing an object for intersection, it is much simpler to test the primitive type from which the object is derived than to test the object itself This is because the primitive lies uniformly sized and positioned in its own local coordinate system, whereas the object lies arbitrarily sized and orientated in world coordinate space

Provided that the ray is correctly transformed from world coordinates to the local coordinate system of the primitive type from which the object is derived, the value for t found from intersecting the transformed ray with the primitive is the same as that found by intersecting the untransformed ray with the object, but is computationally less expensive to calculate The correct transformation is found by using the inverse of the objects 4×4 transformation matrix to transform the ray from world to primitive coordinates In *Micro Trace*, this inverse matrix is calculated for each object before ray tracing commences and stored in the inverse field of the object structure (*table 4 1*)

For a given object, transforming the ray involves transforming both its origin and direction by multiplying them by the object's 4×4 inverse transformation matrix, MI, as follows -

$$\text{transformed ray origin} = [X_0, Y_0, Z_0, 1]\, MI$$

$$\text{transformed ray direction} = [D_x, D_y, D_z, 0]\, MI$$

4.5 Ray Intersection

In its simplest form, the closest object struck by a given ray is determined by testing the ray for intersection with every object in the list and selecting the one that gives the lowest value of t for an intersection (remember t is a measure of the distance of the intersection point from the ray origin) *Section 4 6* however, discusses several optimization techniques that have been implemented which significantly reduce the set of objects that have to be tested to some subset consisting only of those objects with a high probability of being intersected by

the ray, resulting in a substantially reduced image generation time There are four ray-primitive intersection testing functions in *Micro*Trace, one for each of the four primitive types For each primitive type, this test generally involves -

[1] checking if there is a value of t for the ray which gives a potential intersection point with any of the primitive's surfaces

[2] substituting this value for t (if one is found) into the ray equation

[3] checking if the resultant point lies on the corresponding surface of the primitive (and calculating the surface normal at that point if it does)

However, given that we are only interested in the closest point of intersection of the ray with any object, unless the t value from stage [1] is less than that for the closest intersection found in processing the list of objects so far, it can't possibly produce a closer intersection so it is pointless to proceed with stages [2] and [3]

Consequently, each of the four ray-primitive intersection functions outlined in *sections 4 5 1* to *4 5 4* below, in addition to being given a ray to test for intersection, is also given the lowest intersection value of t for the ray with any object in the list tested so far (this value is initially set to some very high value) and the surface normal at the corresponding intersection point Each time a closer intersection is found, the value and surface normal are updated to reflect the new intersection These three entities, the ray to be tested, the closest intersection so far and the surface normal at that intersection are referenced in the following sections as -

ENTITY	DESCRIPTION
(X_0, Y_0, Z_0)	ray origin
(D_x, D_y, D_z)	ray direction
nearest	t value of closest object intersection so far
normal	surface normal at closest intersection so far

4.5.1 Cube Intersection

The cube primitive is a unit cube defined by six surfaces, each one parallel to one of the XYZ axes -

surface equations	bounds test	surface normal
$X = 0$	$0 <= Y, Z <= 1$	(−1, 0, 0)
$X = 1$	$0 <= Y, Z <= 1$	(1, 0, 0)
$Y = 0$	$0 <= X, Z <= 1$	(0, −1, 0)
$Y = 1$	$0 <= X, Z <= 1$	(0, 1, 0)
$Z = 0$	$0 <= X, Y <= 1$	(0, 0, −1)
$Z = 1$	$0 <= X, Y <= 1$	(0, 0, 1)

Finding the closest point of intersection of the cube with a transformed ray involves testing each surface for intersection with the ray For each surface, this involves finding the value of t (if any) for which the ray intersects the corresponding plane If this value is less than "nearest" (see *section 4 5* above) then the exact point of intersection is determined and tested to see if it complies with the bounds tests for the surface If it complies, then an intersection has been found The test for the $Z = 0$ surface is outlined in pseudo-code below, the tests for the remaining surfaces being very similar

Intersection Test For $Z = 0$ Surface

Find t for which ray intersects $Z = 0$ plane -

$$\begin{aligned} Z &= 0 \\ \Rightarrow Zo + tDz &= 0 \\ \Rightarrow t &= -\frac{Z_0}{D_z} \end{aligned} \tag{4 3}$$

Pseudo Code

```
if (D_z is not 0)            /* D_z = 0 ⇒ ray parallel to plane */
 begin
        t = -Z_0 / D_z
        if (t < nearest) and (t > 0)
         begin
                X = X_0 + t D_x   /* calculate point of intersection */
                Y = Y_0 + t D_y
                if (X <= 1) and (X >= 0) and (Y <= 1) and (Y >= 0)
                 begin
                        nearest = t
                        normal = (-1, 0, 0)
                 end
         end
end
```

4.5.2 Sphere Intersection

The sphere primitive is defined as a unit sphere centered at the origin with the following surface equation -

surface equation	bounds test	surface normal
$X^2 + Y^2 + Z^2 = 1$	NONE	(X, Y, Z)

One way of testing if the ray intersects the sphere is to test if the perpendicular distance between the ray and the origin (the centre of the sphere) is less than one (the radius) This simple test will not however give the point of intersection between the ray and the sphere (which is required for calculating the surface normal) Alternatively, substituting the ray equation into the surface equation for the sphere produces a quadratic equation in t which can be solved using a quadratic formula Substituting the solved value for t back into the ray equation then gives the exact point of intersection between the ray and the sphere This latter approach is the one outlined below -

Substituting the ray equation

$$\begin{aligned} X &= X_0 + tD_x \\ Y &= Y_0 + tD_y \\ Z &= Z_0 + tD_z \end{aligned} \tag{4 4}$$

into the sphere equation

$$X^2 + Y^2 + Z^2 = 1$$

gives

$$(X_0 + tD_x)^2 + (Y_0 + tD_y)^2 + (Z_0 + tD_z)^2 = 1$$

Expanding and rearranging gives a quadratic in t which can be solved with the formula

$$t = \frac{B \pm \sqrt{B^2 - AC}}{A}$$

where

$$\begin{aligned} A &= D_x^2 + D_y^2 + D_z^2 \\ B &= D_x X_0 + D_y Y_0 + D_z Z_0 \\ C &= X_0^2 + Y_0^2 + Z_0^2 - 1 \end{aligned} \tag{4 5}$$

The intersection test is then based on the square root term, with the following interpretations -

Value of $B^2 - AC$	Number of intersections	Interpretation of result
zero	1	ray tangental – no intersection
negative	0	complex solution – no intersection
positive	2	two intersections – take closest

4.5.3 Cylinder Intersection

The cylinder primitive consists of a cylinder of unit radius and length, defined by the following three surfaces -

surface equation	bounds test	surface normal
$X^2 + Y^2 = 1$	$0 <= Z <= 1$	$(X, Y, 0)$
$Z = 0$	$X^2 + Y^2 <= 1$	$(0, 0, -1)$
$Z = 1$	$X^2 + Y^2 <= 1$	$(0, 0, 1)$

Intersection Test For $X^2 + Y^2 = 1$ Surface

The test for this surface is derived in the same way as that for the sphere, by substituting the ray equation into the surface equation, with the resultant equation -

$$(X_0 + tD_x)^2 + (Y_0 + tD_y)^2 = 1$$

Expanding and rearranging gives a quadratic in t which again can be solved with the formula

$$t = \frac{B \pm \sqrt{B^2 - AC}}{A}$$

where

$$\begin{aligned} A &= D_x^2 + D_y^2 \\ B &= D_x X_0 + D_y Y_0 \\ C &= X_0^2 + Y_0^2 - 1 \end{aligned} \tag{4 6}$$

As for the sphere, the intersection test is based on the square root term, with the same interpretations

Intersection Test For Other Surfaces

The intersection test for the other two cylinder surfaces is very similar to that of the $Z = 0$ surface for the cube illustrated above, with the exception that the bounds test is slightly different *i e* having calculated a value X and Y, the point at which the ray intersects the plane in question, the test

$(X <= 1)$ and $(X >= 0)$ and $(Y <= 1)$ and $(Y >= 0)$

is replaced by

$$(X^2 + Y^2) <= 1$$

4.5.4 Cone Intersection

The cone primitive consists of a cone of unit height and radius, with the apex at the origin, and is defined by the following two surfaces -

surface equation	bounds test	surface normal
$X^2 + Y^2 - Z^2 = 0$	$0 <= Z <= 1$	$(X, Y, 0)$
$Z = 1$	$X^2 + Y^2 <= 1$	$(X, Y, \sqrt{1 - Z^2})$

Intersection Test For $X^2 + Y^2 - Z^2 = 0$ Surface

The intersection test for this surface, like the sphere, is derived by substituting the ray equation into the surface equation, producing -

$$(X_0 + tD_x)^2 + (Y_0 + tD_y)^2 - (Z_0 + tD_z)^2 = 0$$

Again, expanding and rearranging gives a quadratic in t which as before, can be solved with the formula

$$t = \frac{B \pm \sqrt{B^2 - AC}}{A}$$

where

$$\begin{aligned} A &= D_x^2 + D_y^2 - D_z^2 \\ B &= D_x X_0 + D_y Y_0 - D_z Z_0 \\ C &= X_0^2 + Y_0^2 - Z_0^2 \end{aligned} \tag{4.7}$$

The intersection test for the other surface is identical to that of the $Z = 1$ cylinder surface

4.6 Shadow Rays

Section 2.3.1 described how it was possible to determine if a point was in shadow with respect to a light source by tracing a ray, called a shadow ray, from the point to the light source. If the ray hits any opaque object, the point lies in shadow. *Micro Trace* uses a different set of ray-primitive intersection functions for testing shadow rays in order to take advantage of the fact that the only concern is whether or not the ray strikes any surface between its origin and the light source. The functions are very similar to the ones outlined above and differ only in the respect that as soon as the ray is found to strike any surface of any object, no further surfaces of that object are tested, no surface normal or point of intersection are calculated, and no further objects are tested.

4.7 Optimizations

As outlined in *section 2.4*, 75% of the time taken to ray trace an image is taken up with calculating the intersection of rays with objects. Given that the ray-object intersection calculations have been optimized as far as possible, the only way of reducing this figure is to attempt to reduce the number of ray-object intersection tests. One way of doing this is to try to determine, before testing the ray with the list of objects, those objects that the ray has no chance of hitting. These missed objects can then be excluded from the test, so that the ray is tested for intersection with only a small subset of objects that have a high probability of being hit. The difficulty of such a scheme however, lies in finding a means of easily identifying as many of these missed objects as possible, which will balance the number of such objects detected against the computational cost of detecting them.

The following sections outline four optimizations employed by *Micro*Trace which provide different methods of reducing the number of objects against which a ray must be tested, and have been called :-

- Pixelbuffer
- Extents
- Grid
- Sortlist

The optimizations have been implemented in such a manner as to be completely independent from one another. That is to say, they can be used either individually, or in any combination. Each one is controlled by its own flag variable, which if set means that the optimization is to be employed. The flags in turn are set by calling the appropriate functions from the user interface module (*Micro*Trace sets all four optimizations on by default).

All four optimizations operate only for primary rays and, with the exception of the third, the Pixelbuffer, can operate when either a file and/or a display image output has been requested — the Pixelbuffer can only be used with a display output since it relies on reading pixel values in advance of tracing to determine whether or not a ray should be cast for a given pixel. The effectiveness of the optimizations at reducing rendering time is examined in *section 4.8*, where image generation times for various combinations of the four are tabulated for several

test images

Since all of the optimizations, with the exception of Sortlist rely on the existence of a bounding volume for each of the four primitive object types, before looking at each of the optimizations in detail, a brief description of the bounding volumes used may prove useful

4.7.1 Bounding Volumes

As outlined in *section 2 4*, the majority of optimization techniques currently employed in ray tracing can be broadly classified as either space subdivision or bounding volume techniques One of the reasons for selecting bounding volumes over space subdivision as the basis of most of the *Micro Traces* optimizations was that space subdivision techniques have a tendency to require quite large amounts of memory and, given a memory "budget" of 640K it was felt that bounding volumes would give a better space/performance ratio — regardless of the number of objects in a scene, *Micro Trace* stores a total of only four bounding volumes (one for each of the four primitive object types)

Section 2 4 1 discussed in some detail the issue of bounding volumes where the general idea is to surround each object in a volume that is less costly to test for intersection than the object it encloses Then, only if the ray intersects the bounding volume, is it tested for intersection with the object inside

In *Micro Trace* however, each object is not explicitly surrounded in its own bounding volume Instead, only four bounding volumes exist, one for each of the four primitive types, *fig 4 7* Unlike the primitive types, whose shape and position are not explicitly defined anywhere in the ray tracer (they are implicitly defined in their respective ray-primitive intersection functions), the four bounding volumes are explicitly stored in an array, each as a set of eight XYZ coordinates The bounding volume for a particular object can then be calculated when needed simply by applying its transformation matrix to the eight vertices of the volume of the primitive from which the object is derived

The reason for calculating rather than storing the object bounding volumes is that storing the bounding volume in the object structure would increase the space required to store an object by about 50% In addition, all of the optimizations below make use of object bounding volumes only to extract some initial information about an object Once this information has been extracted and stored in the object structure, there is no further need for the bounding volume.

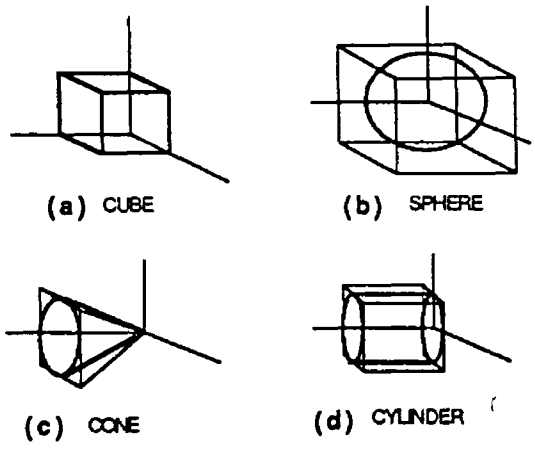


figure 4 7 The four primitive bounding volumes

Consequently, bounding volumes are calculated one at a time for each object before ray tracing commences and are passed to any active optimizations which in turn extract and store any information they require, and are then discarded

It can be seen from *fig 4 7* that the bounding volume for each of the sphere, cylinder and cube, consists of a either a cubic or rectangular block defined by eight coordinates, which form the eight corners of the block In the case of the cone however, the bounding volume is a pyramid, requiring only five vertices for definition For the sake of consistency however, this bounding volume is stored as eight vertices by duplicating the apex coordinate three times This prevents having to treat this volume as a special case and means that all functions which process bounding volumes need be presented with nothing more than a list of eight vertices, regardless of the primitive type to which the bounding volume belongs

4.7.2 Pixelbuffer

For many scenes, a large percentage of pixel values will be set to the background colour *ie* the rays which they spawn do not intersect any objects at all The idea behind this original optimization, which is referred to in *M*icro*T*race as Pixelbuffer, is to provide an easy means of detecting such pixels, so that they can then be automatically set to the background colour without having to generate and trace a ray for them

Firstly, all pixels on the screen are set to a given colour Then, one by one the bounding volume for each object is calculated and drawn on the screen as

six polygons, filled in given colour When all objects have been processed in this way, the result is a screen consisting of pixels that have one of two colours — the initial colour and the colour used to fill the polygons The former are the ones whose rays definitely miss all objects

When ray tracing commences, before casting a ray through a pixel, the colour of the pixel is examined to see if it is the same colour as that used to fill the projected bounding volumes If it isn't, then the pixel is immediately set to the background colour and the next pixel processed

Even though this scheme detects a large percentage of all pixels whose rays do not intersect any object, it cannot always detect all of them since, with the exception of the cube, all of the bounding volumes will contain some void space (*ie* some empty space between the object and the bounding volume) which may project onto pixels that are not otherwise covered by a non-void part of some other volume The percentage of pixels falling into this category however would normally be quite low, but could be further reduced by the use of tighter bounding volumes This however would be at the cost of increasing the overhead of storing, transforming and rendering the bounding volumes, a cost which is unlikely to pay for itself in terms of increased performance of the optimization

COST AND PERFORMANCE The cost of implementing Pixelbuffer consists entirely of the cost of reading each pixel once, which is constant for a given viewport size, and the cost of generating and rendering the bounding volumes, which is proportional to the number objects in the scene The savings obtained from Pixelbuffer on the other hand are directly proportional to the number of empty pixels (ones which do not have the fillcolour after all bounding volumes have been rendered) and is independent of the number of objects in the scene since, given -

$$\begin{aligned} \#\ \text{pixels} &= N \\ \#\ \text{fillcolour pixels} &= F \\ \#\ \text{objects} &= n \end{aligned}$$

the savings are calculated as

$$savings = \frac{(N-F)\,n}{N\,n} = 1 - \frac{F}{N} \qquad (4\,8)$$

While Pixelbuffer provides a means of easily detecting those pixels whose rays will not intersect any object, it has the limitation that pixels failing to fall into this

category must have their rays tested for intersection with all objects in the scene It does however have the advantage of being reasonably effective and easy to implement and does not incur either a large computational or storage overhead

4.7.3 Extents

As outlined in *section 2 4 1*, associating an object with a bounding volume which tightly encloses the object, but is computationally less expensive to test for intersection with a ray, can reduce the cost of ray tracing an image Testing the ray for intersection with the bounding volume however, still requires the use of expensive floating point arithmetic since it is performed in world coordinate space In addition to this, the bounding volume would have to be stored with the object which, as mentioned earlier, would increase the space required to store a single object by about 50%

Extents, which is based on Roth's use of box enclosures [ROTH82] (see *section 3 3 5*), overcomes these two problems however by using a 2D "bounding volume" called an extent, which is essentially a rectangle in screen space that encloses the object, *section 2 4 1* The first problem, that of having to use floating point arithmetic, is thus overcome since testing of extents is done in integer screen space and the second, that of storage, by virtue of the fact that an extent requires storage for just four integers, which compares very favorably with the twenty four floating point numbers required for a bounding volume In fact, in *Microsoft C* version *5 1*, where an integer occupies 2 bytes, and a long float 8, an extent requires only 4% of the storage required for a bounding volume (8 bytes compared to 192)

Object extents are calculated for each object before ray tracing commences by generating a bounding volume for each object and finding its minimum and maximum X and Y coordinates, which are then projected onto screen coordinates and stored as four integers in the *xmin, xmax, ymin*, and *ymax* fields respectively of the structure for the object (*table 4 1*) These integer coordinates $(xmin, ymin)$ and $(xmax, ymax)$ form the lower left and upper right corners of the screen extent of the object

When ray tracing, before testing a ray for intersection with a particular object, the pixel that spawned the ray is tested against the object's screen extent Only if the pixel lies inside the extent, is the ray tested for intersection with the object

COST AND PERFORMANCE The cost of using Extents amounts to the cost of generating the extent for each object, plus the cost of testing a pixel against the extent each time the object is tested for intersection with a ray. The former consists of the cost of generating the bounding volume for each object (which is divided among all active optimization schemes) as well as the cost of finding and projecting the minimum and maximum X and Y coordinates. The test to see if a pixel (x, y) lies inside an extent requires, in the worst case, 4 integer comparisons (since all of the four conditions are "and" conditions, as soon as one fails, the overall condition fails)

$$(x <= x_{max}) \text{ and } (x >= x_{min}) \text{ and } (y <= y_{max}) \text{ and } (y >= y_{min})$$

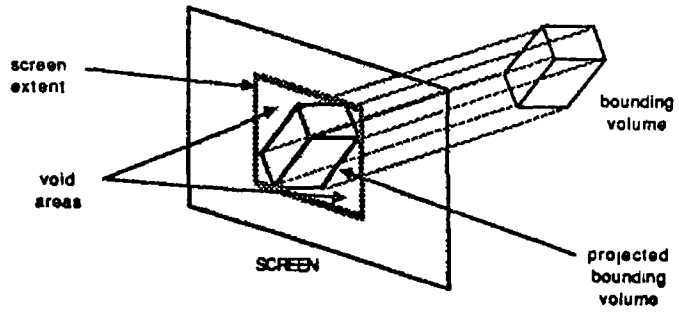
In the worst possible case, where the pixel for every ray lies inside the extent for every object, extents become a liability rather than an asset since testing the pixel with an objects extent becomes superfluous and merely adds to the cost of testing the ray with the object. Even in such rare cases however, the liability would be minimal since the cost of the pixel-extent test is negligible in comparison to that of the ray-object test. Optimal savings on the other hand, would occur for a scene containing a large number of objects with as little overlapping of extents as possible. In the best possible case, where there is no overlapping at all, each primary ray would then have to be intersected with (at most) a single object from the entire list.

While the "ray intersects extent" test is much cheaper to perform, both from a computational and a storage point of view, than the "ray intersects bounding volume" test, the fact that it is performed in screen space means that it can only be applied to primary rays, since these are the only rays that are constrained to pass through the screen. The bounding volume test on the other hand, is performed in object space and hence can be applied to both primary and secondary rays. However, there is nothing to prevent both schemes being employed simultaneously *ie.* to use screen space extents for testing primary rays, and to use object space bounding volumes for secondary rays — a possibility outlined in *section 5.2.1.*

The limitation on the use of extents to primary rays is not of major significance to *Micro Trace* at present since, by virtue of the fact that it currently incorporates just a single light source, and that the only secondary rays traced are shadow rays, the number of primary rays cast will always be greater than the number of secondary rays (except in the rare case where all primary rays intersect an object, when the two will be equal). However, should *Micro Trace*, at some point in the future, be upgraded to incorporate multiple light sources and/or transparent objects, it is quite likely that the number of secondary rays

will exceed that of primary ones and secondary ray optimizations would then be a significant factor in its performance Such upgrades and secondary ray optimizations are discussed in *section 5 2*

Another limitation on the usefulness of extents is that, depending on the shape and orientation of the object, they can sometimes contain large void areas, which reduces their effectiveness, *fig 4 8* One way of reducing this void area would be to use a polygonal extent instead of a rectangular one Such an extent could enclose the projected volumes without any void area since all four bounding volumes in *Micro*Trace consist of polygons and so will always form a polygon when projected onto the screen The cost of generating polygonal extents however would be greater than that for rectangular ones, as would the cost of storage and testing



Screen extents can sometimes contain large void areas

figure 4 8

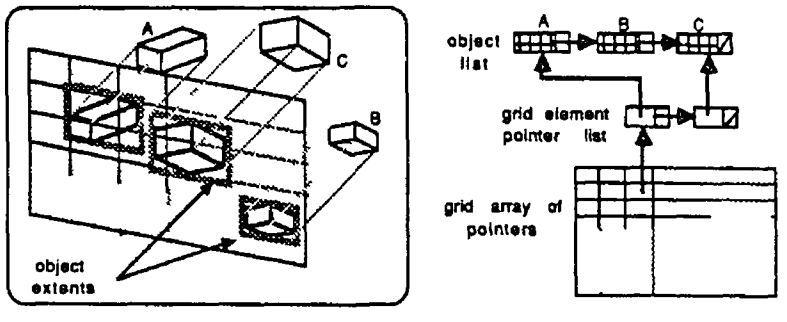
4.7.4 Grid

While the Extents optimization outlined in the previous section provides a quick and easy test to see if a primary ray stands a good chance of hitting an object, the complete list of objects is still processed for every ray The Grid optimization on the other hand provides a means of supplying a list of objects for each ray that is a subset of the entire list of objects, consisting only of those objects with a high probability of being intersected The optimization is based on ideas by Gervautz [GERV86] for partitioning screen space in order to create temporary object trees (*section 3 4 4*) and by Arnaldi [ARNA87] for generating 3D cells for space subdivision (*section 3 4 5*)

The optimization resembles something of a space subdivision technique (*section 2 4 2*) in two dimensional screen space as opposed to three dimensional ob-

ject space The idea is to impose a rectangular grid on the screen and to associate with each rectangle a list of objects that are partially/completely contained in that rectangle Testing a ray for intersection with the scene then involves determining the rectangle to which the pixel spawning the ray belongs, and testing the ray for intersection with the associated list of objects for that rectangle Since, as mentioned earlier, all four optimizations can be used either together or independently, any of the other optimizations can be used to speed up the testing of the ray with this abbreviated list

Micro Trace uses a fixed grid size of 20 cells horizontally by 20 cells vertically regardless of the size of the screen Consequently, the larger the screen, the greater the number of pixels associated with each cell The grid is implemented as a $20x20$ array of pointers to linked lists, with each element of a linked list containing a single pointer to an object whose extent overlaps the cell, *fig 4 9* While storing the grid as an array means that the number of cells is fixed (changing the number requires re-defining the array dimensions in the source code and recompiling) it does have the advantage of providing fast and easy access to the object pointer list for a given pixel, which simply involves using the pixels coordinates as an index into the array Alternative structures which provide a more flexible approach for implementing the grid are discussed in *section 5 2 1*



Implementation of GRID using array of pointers to linked lists

figure 4 9

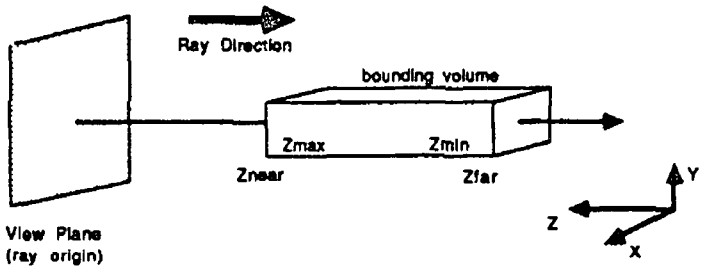
The grid is generated by scanning the list of objects once for each cell, generating the linked list of pointers, and then setting the corresponding array entry to point to the list Cells that do not contain any objects have a *NULL* pointer in their corresponding array entry In determining whether or not an object overlaps a cell, the objects screen extent is used instead of its bounding volume Although this will sometimes give a less accurate approximation as to whether an object overlaps a cell than if the the objects screen projected

bounding volume were used (extents generally contain a larger percentage of void space), the former is a less expensive test to perform In addition, since the Grid optimization would normally be used in conjunction with the Extents optimization, the computational expense of generating the grid is further reduced as object extents will already have been generated

4.7.5 Sortlist

Suppose that, instead of having to test a ray for intersection with with a list of objects in random order, the objects in the list were presented in the same order in which they are encountered by the ray The first intersection of the ray with such a list would then be guaranteed to be the closest one, so as soon as an intersection is found, the remaining objects in the list need not be tested This is the general idea behind another original optimization employed by *Micro Trace*, called Sortlist

Before ray tracing commences, bounding volumes are calculated for each object The closest and furthest Z coordinates of each from the origin of the ray, are calculated and stored in the *znear* and *zfar* fields respectively of the object structure (see *table 4 1*) The terms closest and furthest are used in place of minimum and maximum since, from *fig 4 10*, depending on the direction of the primary ray, Z_{min} could be either the closest or the furthest from the ray origin

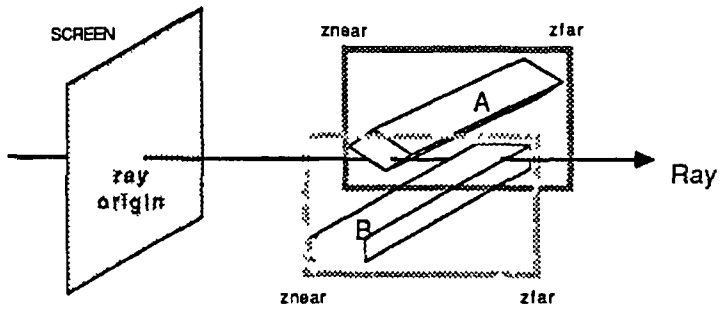


A case where Znear is equal to Zmax instead of Zmin

figure 4 10

The linked list of objects presented to *Micro Trace* is then sorted into ascending order of *znear* coordinates by rearranging the links in the list, a function which is performed by an efficient sorting algorithm for linked lists that requires of the order of $N \log N$ comparisons (where N is the number of objects in the list) [ERDE89]

Since this list represents only an approximation to the order in which a ray will encounter the objects, as illustrated in *fig 4 11*, it is unsafe to cease testing objects in the list as soon as an intersection is found However, given Z_c, the $Z-$coordinate of the closest intersection found in processing the list so far, it is always safe to cease testing as soon as an object is encountered whose *znear* coordinate lies further from the ray origin than Z_c



Even though object B lies closer to the ray origin than A the ray intersects object A at a closer point

figure 4 11

COST AND PERFORMANCE The cost of implementing the sorting optimization consists of the cost of initially generating the sorted list of objects, plus the cost per ray of testing the *znear* of each object with Z_c This latter cost in turn will depend on how soon a safe exit point is reached in the list of objects The conditions for optimal performance of Sortlist would be a scene where there is little or no overlapping of objects along the $Z-$axis (overlapping on X and Y axes would have no effect) This is in contrast to the Extents optimization, where overlapping along the $Z-$axis has no affect on performance but overlapping on the X and Y axes does

4.8 Presentation of Results

This section evaluates the performance of each of the four optimizations outlined above by presenting and analysing tracing times and other statistics for two test images generated by *Micro*Trace Photographs of the two images, a scene of snooker balls and a chemical lattice, are shown in *figs 4 12* and *4 13*, along with coloured diagrams illustrating the grid cells and object extents (*figs 4 14* and *4 15*)

4.8.1 The Test Images

Snooker Balls *Figure 4 12* shows an image of a set of snooker balls with the viewer situated behind and above the pack of red balls, looking down the table The scene consists of 253,440 pixels and contains 21 objects, each one a sphere The degree to which object extents overlap each other as well as their distribution among the grid cells (two factors which affect the efficiency of the Extent and Grid optimizations) can be seen from *fig 4 14a*, which illustrates the cells of the grid and the object screen extents[2] as they would appear to *Micro Trace* In addition, the degree of overlapping of the *znear* and *zfar* coordinates of the object bounding volumes, a factor which affects the efficiency of Sortlist, is illustrated in *fig 4 14b*, which shows the bounding volumes when orthographically projected onto the xz-plane *ie* when viewed from above

Lattice *Figure 4 13* shows the lattice scene, which consists of 54 objects — 27 spheres and 27 cylinders The number of pixels for this scene (223,680) is different from that of the snooker balls by virtue of the fact that the program which generated the scene automatically calculates a window on the view plane that is just sufficient to enclose the scene and adjusts the specified viewport to maintain the correct window/viewport ratio As for the snooker scene, *fig 4 15a* illustrates the grid cells and object extents and *fig 4 15b* shows a plan view of the scene

4.8.2 Explaination Of Terms

The results of *tables 4 3* and *4 4* were gathered by tracing each of the lattice and snooker images six times — once with no optimizations active, once with all optimizations active and once with one of each of the four optimizations active The derivation and meaning of each statistic is outlined in *table 4 2*

The first two statistics of each table, "rays traced" and "rays intersected" are really only relevant to cases where the Pixelbuffer optimization is active The value of "rays generated" would represent the number of pixels that had the fillcolour after the Pixelbuffer preprocess (*section 4 7 2*), while "rays intersected" would represent the percentage of those pixels whose rays intersected an object and therefore provides an indication of the void space contained in the projected bounding volumes — the *higher* the percentage, the *lower* the amount of void space

[2]The extents are outlined in red and filled in yellow for clarity

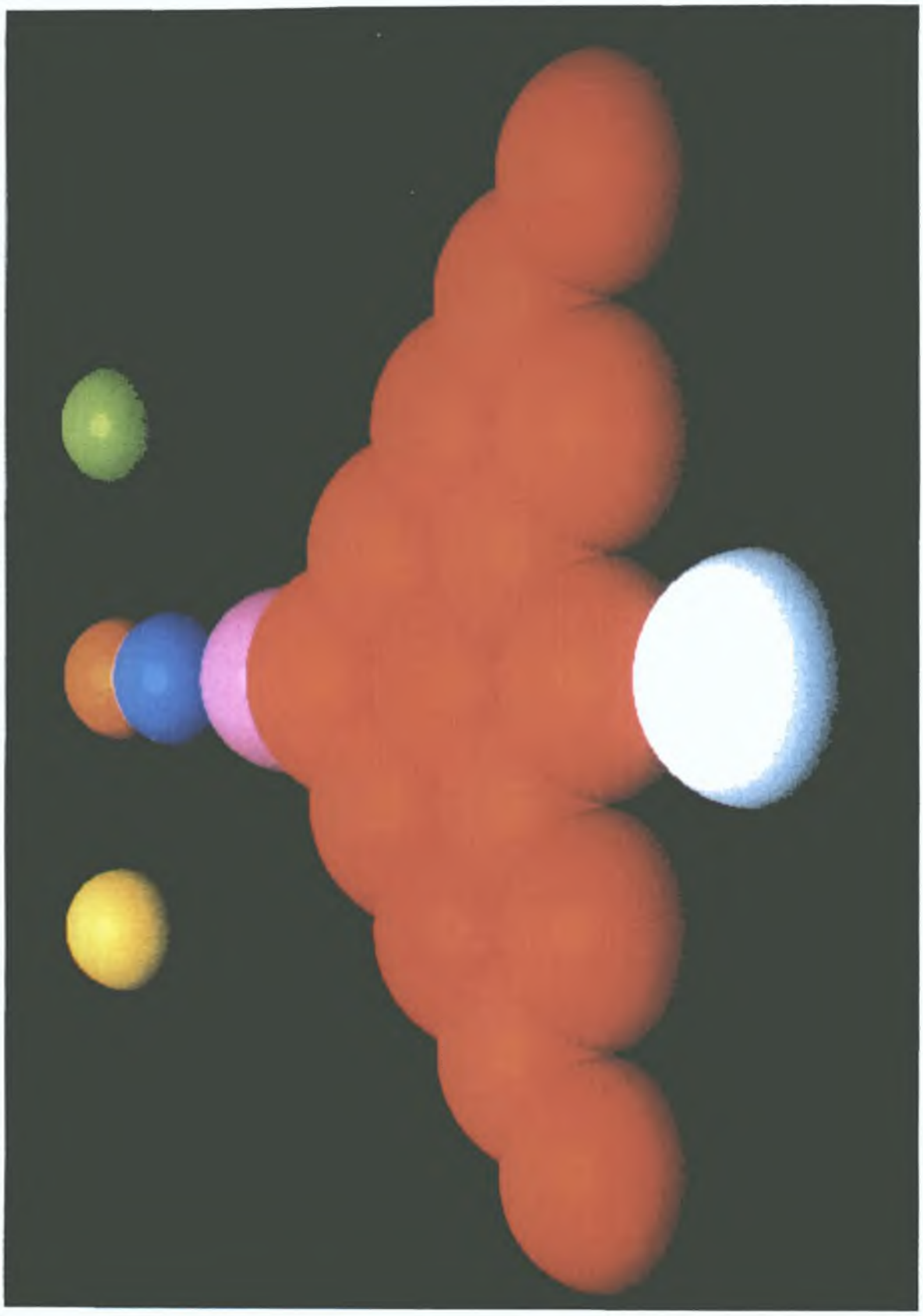


Figure 4.12: Snooker scene generated by MicroTrace

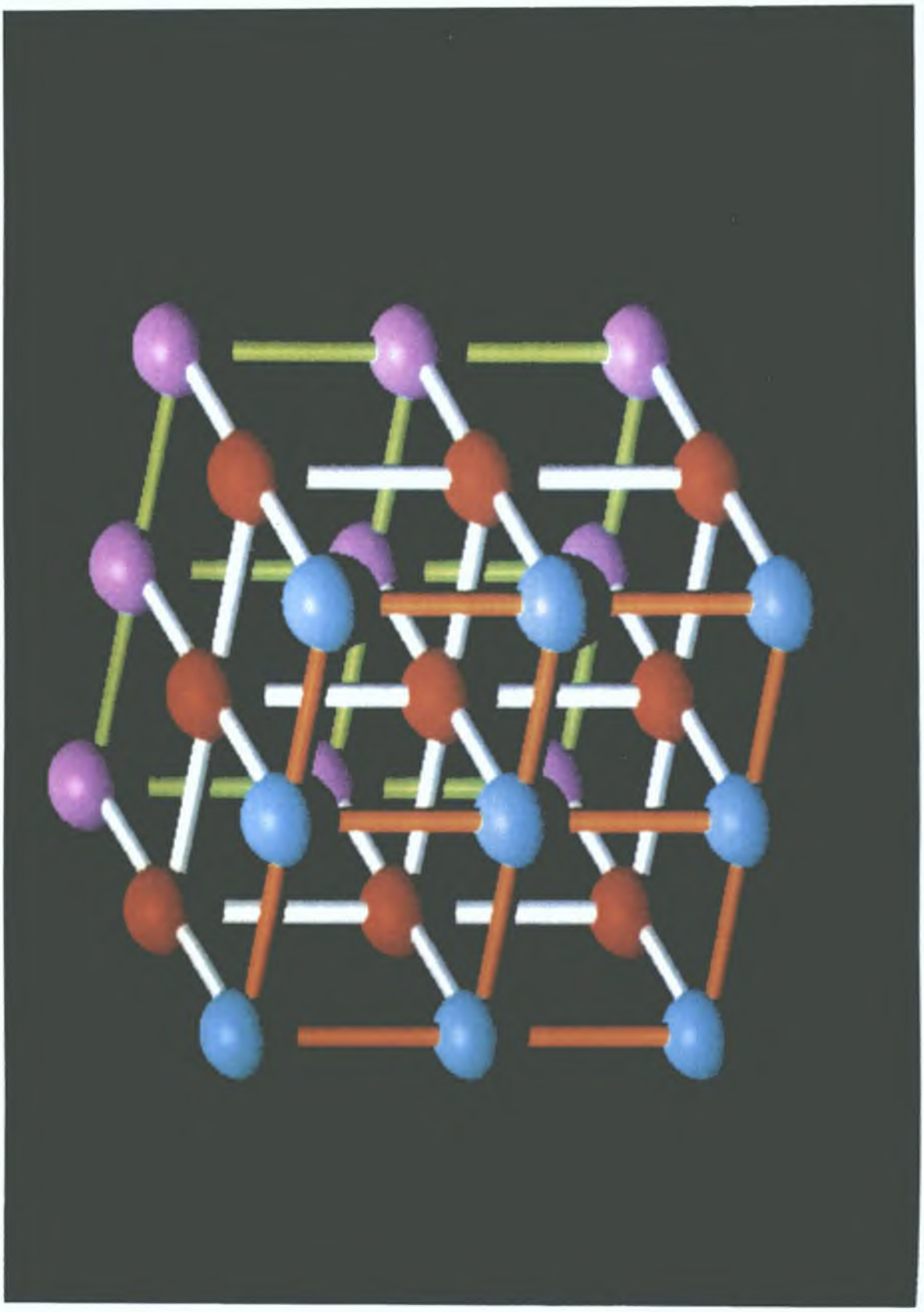


Figure 4.13: Lattice scene generated by MicroTrace

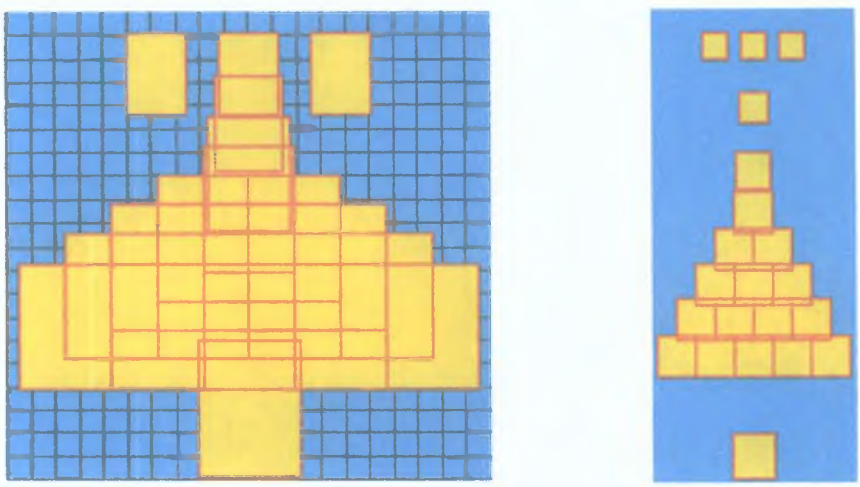
STATISTIC	DESCRIPTION
rays traced	• % of pixels for which a ray was generated
rays intersected	• % of generated rays which intersected some object
R–O Tests -	
per ray	• Average no of ray-object tests per ray
total	• Total no of ray-object tests in *millions*
% reduction	• Reduction in required no of ray-object tests as a % of that required to test *every* ray with *every* object
Time -	
preprocess	• Time to calculate inverse transformation matrices, sort object list, calculate extents *etc*
Total	• Ray tracing time + preprocess time (hh mm ss)
% N O time	• Total time as a % of non-optimized time

Table 4 2 Description of results terms

In relation to ray-objects tests, the tests per ray, total tests and percentage reduction, illustrate from different perspectives, the overall effectiveness of the active optimization(s) in reducing the number of required ray-objects tests which, as can be seen from the tables, has the greatest bearing on speeding up tracing times Also in relation to ray-object tests, the percentage of tests resulting in an intersection, "% hits", gives an indication of the efficiency of the Extents and Grid optimizations (*sections 4 7 3* and *4 7 4*) at limiting ray intersection tests to objects with a high probability of being hit, and of the effectiveness of the sorting of the Sortlist optimization (*section 4 7 5*)

4.8.3 Discussion Of Results

From a glance at *tables 4 2* and *4 2*, which list the statictics for each optimization in order of decreasing rendering time, it is clear that Extents provides the greatest time savings for both images and that no optimization is slower than the unoptimized time Note however that the order of optimizations in both tables is different — Grid is the second slowest optimization for the lattice whereas

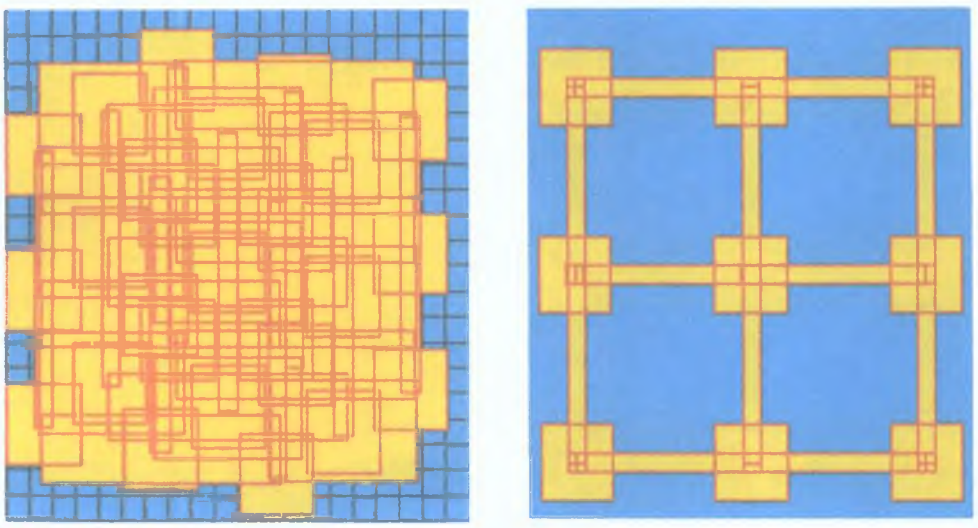


Grid & Extents (left) and plan view (right) for SNOOKER image

Figure 4.14 (a) and (b)

SNOOKER BALLS ⟶ 21 objects, 253440 pixels						
STATISTIC	**Optimizations**					
	none	Sl	Pb	Gd	Ex	All
rays traced	100.00%	100.00%	49.57%	100.00%	100.00%	49.57%
rays intersected	36.99%	36.99%	74.62%	36.99%	36.99%	74.62%
R-O Tests :-						
per ray	21.00	16.85	21.00	8.77	0.98	1.27
% hits	10.17%	9.18%	5.66%	13.56%	60.13%	58.83%
total	5.30 M	4.27 M	2.64 M	2.22 M	0.25 M	0.16 M
% reduction	0.00%	19.76%	50.43%	58.24%	95.34%	97.01%
Time :-						
preprocess	0 sec	1 sec	7 sec	2 sec	0 sec	7 sec
Total	4:35:06	3:48:18	2:22:26	2:02:55	00:30:51	00:22:04
% N.O. time	100.00%	82.99%	51.76%	44.68%	11.21%	8.02%

Table 4.3: Snooker Scene Statistics.



Grid & Extents (left) and plan view (right) for LATTICE image

Figure 4.15 (a) and (b)

LATTICE ⟶ 54 objects, 223680 pixels						
STATISTIC	**Optimizations**					
	none	Sl	Gd	Pb	Ex	All
rays traced	100.00%	100.00%	100.00%	63.52%	100.00%	63.53%
rays intersected	41.58%	41.58%	41.58%	65.46%	41.58%	65.46%
R–O Tests :-						
per ray	54.00	44.99	35.07	54.00	2.09	2.43
% hits	4.54%	4.03%	3.76%	1.86%	30.52%	34.98%
total	12.07 M	10.06 M	7.84 M	7.67 M	0.47 M	0.34 M
% reduction	0.00%	16.69%	35.06%	36.48%	96.13%	97.14%
Time :-						
preprocess	1 sec	2 sec	2 sec	16 sec	2 sec	17 sec
Total	12:35:16	10:53:38	8:07:39	7:59:17	00:55:43	00:41:50
% N.O. time	100.00%	86.54%	64.57%	63.46%	7.37%	5.53%

Table 4.4: Lattice Scene Statistics.

Pixelbuffer is for the snooker balls In both cases however, Sortlist is the slowest optimization, though it gives a greater saving for the snooker balls than for the lattice These and other interesting aspects of the results are further analyzed below under the headings of their respective optimizations

Extents From both results tables it is clear that Extents gives a far greater reduction in tracing time than any single other optimization — the time for the snooker balls being reduced to just over 11% of the unoptimized time and that of the lattice to just over 7% It can also be seen that the "% hits" for the snooker balls, at 60 13%, is almost twice that for the lattice, 30 52% The lower figure for the lattice is probably a result of the size and orientation of the cylinders, which causes their extents to contain a larger amount of void space and consequently results in a greater number of non-intersecting rays having to be tested for these extents

It is interesting to note however that in spite of this fact, and the fact that there is greater overlapping of extents (compare *fig 4 14a* with *4 15a*), Extents produces a greater saving in rendering time for the lattice scene This is probably due to the slightly larger reduction in ray-objects tests for this scene, 96 13% compared to 95 34%, and the larger number of objects (54 against 21) though the issue is clouded somewhat by the fact that the cylinders in lattice scene have a more expensive ray-intersection test than the spheres so figures for reductions in ray-object tests are not as straight forward as for the snooker scene, where all objects have the same test cost

Pixelbuffer In *section 4 7 2* it was outlined how the saving from Pixelbuffer is related only to the number of fillcolour pixels and is independent of the number of objects in the scene This fact is confirmed from the figures in the results tables where, for both images, the sum of the "rays traced" percentage and "% reduction" is 100, indicating the relationship of *equation 4 8* In addition, the rendering time, expressed as a percentage of the unoptimized rendering time, is almost proportional to the "rays traced" figure which, as outlined above, indicates the percentage of pixels that had the fillcolour after the Pixelbuffer preprocess A comparison of the "rays intersected" statistic for both images shows that a higher percentage of rays were intersected in the snooker scene, indicating that the projected void area (on the view plane) of the bounding volumes for this scene were, on average, less than those in the lattice scene Note that the significantly longer preprocess times for this optimization is, in both images, a result of the time taken to draw the filled polygons of the bounding volumes on screen

Grid The performance of the Grid optimization can be seen from the tables to be better for the snooker than for the lattice scene A look a *fig 4 14a* and

4 15a shows that the reason is probably due to the greater degree of overlapping in the object extents for the lattice scene, resulting in a larger number of objects being associated with many cells In addition this scene has a much smaller percentage of empty cells It can be seen from the tables too, that the number of ray-object intersection tests per ray is greater, as a percentage of the number of objects in the scene, for the lattice image — again, this is probably a result of the greater overlapping and dispersion, over the grid area, of the object extents These observations, together with the fact that the percentage of ray-object tests that prove positive ("% hits") is significantly lower for the lattice seems to suggest that smaller cells, or a distribution of objects over some sort of hierarchical cell structure (*section 5 2 1*) might prove more benefical for the lattice scene

Sortlist In both images, Sortlist produces the least reductions in rendering times of any of the optimizations but, in spite of this, still manages to render the snooker image in just under half the unoptimized time, with a figure of just over 86% for the lattice A look at *fig 4 14b* shows that there is little overlapping of *znear* and *zfar* coordinates for the snooker scene In examining *fig 4 15b* however, which appears to show little overlapping, it must be remembered that there are in fact two similar tiers of objects directly below the visible one shown So, while overlapping *within* the three horizontal tiers is small, for the scene as a whole it is significantly greater than for the snooker scene — a fact indicated not only by a comparison of percentage rendering times but also by a lower percentage of ray hits and a lower percentage reduction of ray object intersection tests

Overall From the tables, it is celar that the four optimizations, when employed simultaneously, significantly reduce rendering times for both images — the lattice scene takes only 5 53% of the unoptimized rendering time and the snooker scene just 8 02% A look at the "% hits" statistic however, which is 34 4% for the lattice and 58 83% for the snooker balls, combined with the fact that over 90% of the above reductions in rendering times results from the Extents optimization, reveals that there is room for yet further optimization, particularly in the other three optimizations Ways in which this can be achieved, along with suggestions for additional optimizations, are outlined in *section 5 2 1*

4.8.4 Results For Other Machines

During the course of this research, microcomputers with much greater calculating capabilities became available From the ray tracing times[3] and technical

[3]The times given are tracing times for both images using the Grid, Extents and Sortlist optimizations, with no screen of file output

specifications for these machines (*table 4 5*) it can be seen that up to a *ten-fold* speed increase on the AT was achieved by some This is only lead one to imagine where ray tracing will lead in the future, as yet more powerful and faster machines become available — a prospect discussed in *section 5 3*

Machine	Specification		Tracing Time	
			Snooker	Lattice
	Processor	co-processor	min sec	min sec
IBM AT	80286 6*MHz*	80287	22 10	41 55
IBM PS-2	80386 16*MHz*	80387	3 46	6 31
Sun 386i	80386 25*MHz*	80387	2 20	4 08
Olivetti 386	80386 25*MHz*	80387	2 04	3 34

Table 4 3 Tracing Times For Various Machines

Chapter 5

Conclusions & Further Work

5.1 Conclusions

It is clear from the results of *section 4.8* that current microcomputers are well capable of handling the massive computation involved in generating ray traced images which, only a few years earlier, would have been the sole domain of mainframe and super computers. The ever increasing speed and power of these microcomputers (in the duration of this research, a *ten-fold* speed increase occurred, *table 4.5*) will allow more complex and realistic images to be generated, while their decreasing cost will ensure a wider base of use and application of the area realistic image synthesis, and consequently of ray tracing. While it is difficult to make future predictions regarding an area as rapidly expanding and developing as computer graphics, it does not seem unreasonable to expect that in the foreseeable future, the generation of realistic images will be as available and standard a use for microcomputers as desktop publishing is today.

5.2 Future Work

The following sections outline the incorporation of several additional features to *Micro*Trace that will further enhance and extend its rendering and trace-speed capabilities. The enhancements are discussed either in *section 5.2.1* or *section 5.2.2*, depending on whether or not their implementation would require modification of the ray tracing algorithm on which the raytracer is based.

5.2.1 Enhancing *MicroTrace*

This section outlines several recommendations for further enhancing the speed and performance of *MicroTrace*, which do not require modification of its ray tracing algorithm or alter its nature as a primitive instancing rendering system

OPTIMIZING THE OPTIMIZATIONS While it is clear from *section 4 8* that the four optimizations employed by *MicroTrace*, *Pixelbuffer, Extents, Grid* and *Sortlist*, significantly improve rendering times, further optimization should still be possible from finer tuning of four, as well as from the addition of new optimizations

Extents The Extents optimization (*section 4 7 3*), which at present stores each object's extent in the object structure itself, could be modified to incorporate hierarchical extent information either in a separate structure or in a modified object list structure The addition of a second pointer field to the object structure of *table 4 1* would allow for a two dimensionally structured object list which could be fashioned into some form of hierarchy either by the user or automatically, as a preprocess to tracing

Pixelbuffer The Pixelbuffer optimization (*section 4 7 2*) which at present fills each object bounding volume in a single "fill" colour, could be modified to a simplified item buffer of the kind implemented by Weghorst [WEGH84] (*section 2 4 5*) by filling each volume in a unique colour If, as in the Sortlist optimization, the bounding volumes were sorted on the basis of their *znear* co-ordinates before being filled, bounding volumes further from the ray would be overwritten by closer obscuring ones If each bounding volume could then be filled in a unique colour, the colour of a pixel would be a direct indication of the bounding volume closest to the ray origin for that pixel

While there is still no guarantee that the object enclosed in this bounding volume is the closest one intersected by the ray, some objects can still be eliminated from the search, since only those behind the bounding volume mapped onto the pixel have to be tested In a case where there were more than 255 primitives however, bounding volumes would have to share colours so, each pixel colour, instead of corresponding to a single bounding volume, would correspond to the group associated with that colour Such a group colour scheme could be used in conjunction with the hierarchy scheme above to directly identify various groups of the hierarchy without having to search the hierarchical structure

Grid Increased performance from the Grid optimization (*section 4 7 4*) could probably be achieved by using a more flexible structure than the rigidly imposed

20×20 cell array structure currently implemented This could involve either allowing a dynamically determined number of cells horizontally and vertically by implementing the grid as a two-dimensional linked list structure, or more flexible still, a quadtree structure generated along the lines of that used by Gervautz [GERV86] to generate temporary object trees (*section 3 4 4*)

ADDITIONAL PRIMITIVES The set of four primitive types currently implemented by *Micro Trace* (cube, sphere, cylinder, cone) could be augmented by the addition of several new primitive types The addition of a new primitive type requires minor code modification and primarily involves the inclusion of a function that will test a ray for intersection with the primitive, returning a t value and surface normal if it does The addition of a *torus* primitive for example, defined by -

surface equation	bounds test
$(X^2 + Y^2 + Z^2 + 1 - r^2)^2$ $-4(X^2 + Y^2) = 0$	NONE

would involve the inclusion of a function that could solve the fourth order equation in t produced by substituting the ray equation into the above surface equation, along with two minor modifications — the inclusion of the new primitive type's name in the list of available primitives, and a statement in the ray intersection function to call the newly added ray-torus intersection function whenever a torus is encountered in the object list

ANTIALIASING As outlined in *section 2 3 3*, aliasing is a "noise" effect that can often manifest itself in graphics images and have a degrading affect on image quality While the incorporation of any of the antialiasing techniques described in that section would provide *Micro Trace* with some measure of protection against the damaging effects of aliasing, given the microcomputer environment and memory limitation on which *Micro Trace* currently runs, the method described by *figure 2 9* would seem the most appropriate, since the additional antialiasing rays are shared among adjacent pixels

NEW OPTIMIZATIONS At present, all four optimizations employed by *Micro Trace* are configured to optimize for primary rays since, at present, the only secondary rays traced are shadow rays However, the inclusion of some form of optimization that would be applicable to secondary rays would benefit tracing times for images where a user has specified the incorporation of shadows into the

image One method would be to explicitly store each object's bounding volume in the object structure, allowing secondary rays to be directly tested against the bounding volume to determine if the object inside needs to be tested As outlined in *section 4 7 1* however, storing the bounding volume in the object structure would result in a 50% increase its size

A more economical method would be to store a 3-dimensional rectangular parallelpiped that encloses the bounding volume This would require storage for just two XYZ coordinates, as opposed to eight for the bounding volume, but would contain a greater void area, resulting in a larger percentage of rays that do not intersect the object, intersecting the bounding volume Alternatively, spherical bounding volumes could be used and stored as just two floating point values, a center and radius Another alternative would be the implementation of a space subdivision scheme along the lines of *section 2 4 2* — a uniform space subdivision would probably be the most straight forward

5.2.2 Extending *Micro*Trace

This section outlines two extensions to *Micro*Trace that would require some modification of the ray tracing algorithm for their implementation The first, the incorporation of reflection and refraction in screen images would further improve the realism of scenes traced by *Micro*Trace, while the second, the extension of the object representation scheme from a primitive instancing to a CSG representation, would enlarge the range and complexity of solids that could be handled

REFLECTION & REFRACTION The method currently employed by *Micro*Trace to calculate the colour and intensity of a pixel, is to fire a single ray through the pixel into the scene and apply Phong's lighting equation [BUIT75] of *section 1 7 2* at the intersection point of the closest ray-object intersection While accounting, to a reasonably accurate degree, for the ambient, diffuse and specular reflection from the specified point on the object, the equation is applied devoid of the objects context in the overall scene Consequently, the reflection of one object in another, or the incorporation of transparent objects cannot be modelled in the final screen image Whitted's extended lighting model [WHIT80] however provides a method of accumulating the global illumination information necessary to account for these effects As outlined in *section 2 3 2*, the model proposes that on striking an object, a ray be divided into its specularly reflected and transmitted rays These rays in turn are recursively traced to see if they strike any objects, allowing the illumination information for the original ray to be built up in the form of a binary tree

The incorporation of this model into *Micro*Trace would primarily involve making the ray-scene intersection function recursive (the ray-primitive intersection functions would remain unaltered) so that on striking an object the reflected and refracted rays are recursively traced and the intersection information placed in an appropriate binary tree structure. The colour and intensity of a pixel would then be determined by applying Phong's lighting equation to each node of the tree, starting with the leaf nodes and working recursively up to the root node, which would represent the final colour of the pixel. The addition of a *transmission coefficient* and a *refractive index* field to the object structure would also be required for calculation of the direction of a refracted ray through a transparent object.

CSG REPRESENTATION At present, *Micro*Trace provides a set of four primitive object types with which a user can build up a scene through the application of scaling, rotational and translational operations on the primitive types, using the matrix transformation operations in the user interface module. While these transformations provide a means of generating a variety of shapes from just four basic ones (*e.g.* a rectangular block of arbitrary dimensions can be generated by appropriately scaling the cube primitive), the ability to combine solids using the union, difference and intersection boolean operations of *section 3.2.1* would significantly increase the range and complexity of solids that could be generated.

Extending the current primitive instancing representation scheme to a CSG representation would require either a binary tree or DAG solid description scheme in place of the current linked list description. In addition, the ray-solid intersection function, which at present involves a linear search through the linked list, would have to recursively search the binary tree (or DAG) description, starting with the leaf nodes and working recursively up to the root node (*section 3.3*), combining the ray classifications at each node as outlined in *section 3.3.4*. In order that the classifications could be correctly combined, the ray-primitive intersection functions would also have to be modified to return a list of *all* values of t for which the ray intersects the primitive — they currently return only the closest intersection point.

5.3 Ray Tracing — The Future

Ray tracing, despite its large computational overhead, and its labelling by some as a brute force method, looks set to continue as the dominant force in the synthesis of realistic computer images. At the same time, the generation of photorealistic computer images is no longer of purely academic interest but is gradually moving

out into the everyday world, a fact that is emphasized by the recent announcement of *Renderman* by Pixar Inc which is a new proposed standard interface between three-dimensional geometric modelling systems and photorealistic rendering systems [APOD89] The separation of these traditionally integrated modelling and rendering operations, together with a standard interface between the two, should mean that in the future it will be possible to independently select modelling and rendering systems thereby making it simpler to upgrade a system as more realistic renderers become available

Ray tracing is probably capable of dealing with more of the many issues of realistic image synthesis incorporated into *Renderman*, such as motion blurr, antialiasing, shadows, texture mapping and programmable shading languages, than any other advanced rendering system currently available In fact, a variation of ray tracing, known as distributed ray tracing and outlined in *section 2 5 1*, has already been developed and used by Porter [PORT84] to generate one of the first computer images to pass as a photograph

With the major advances in the speed and complexity of graphics and microprocessor hardware in the past decade, this state of the art in computer graphics research is no longer the confine of those fortunate enough to have access to large, expensive mainframes but is rapidly becoming more widely available to the large base of microcomputer users — a fact that is reinforced by the recent appearance of ray tracing packages for microcomputers such as the Acorn *Archimedes* and the Commodore *Amiga*

Already, the calculating power of many microcomputers is approaching and even exceeding that of vastly more expensive mainframes and seems set to increase still further in the near future Intel for example have recently announced the 80860 40 *MHz* RISC processor (Reduced Instruction Set Computer) with a built in maths coprocessor capable of 17 *million* floating point operations per second, giving it about 40 times the computing power of the machine on which this research was carried out [HENN89] At the same time, graphics displays with resolutions of 800×600 pixels that can display up to 256 simultaneous colours from a palett of over 16 *million* are being fitted as standard to these increasingly powerful machines These two developments should see the advent of better, faster and more widely available ray tracing programs in the future, thereby giving it a wider base of application In fact it would not be over ambitious to say that packages for the synthesis of realistic images will be as available and varied on future microcomputers as those for desk top publishing are today

In parallel with these hardware developments, recent ray tracing research has resulted in algorithms capable of taking full advantage of the powerful micro-

computers of the future For example, the recent advent of microcomputers such as the Atari ABAQ, which has not just one, but up to 12, powerful transputer processors executing in parallel, presents exciting possibilities for the implementation of some of the parallel ray tracing algorithms outlined in *section 2 4 4* and seems to be a promising avenue in the ultimate quest for real-time, or near real-time ray tracing

Another interesting possibility is that of a hardware implementation of ray-object intersection tests Pulleyblank examines just such a possibility in his paper on the feasibility of a VLSI chip for ray-tracing bicubic patches His estimates indicate that such a chip could calculate ray-patch intersections at a rate of one every 15ms, or about 67 per second Given the fact that rays can be traced independently of each other, several of these chips operating in parallel would constitute a very powerful ray-tracing engine In fact, judging by the current trend in hardware implementation of "conventional" graphics algorithms such as line drawing, polygon shading and z-buffering, graphics cards with in built hardware facilities for ray tracing would not seem too much of an impossibility in the future

Appendix A

Source Code

```
/* FUNCTION HEADER FILE
   ================================================================
  |                                                                |
  | This  file  contains the function prototype declarations for all |
  | the functions of the raytracer, from the following modules  -    |
  |                                                                |
  |            RTRACE C              SHADE C                       |
  |            RAYINTER C            PGADPEND C                    |
  |            OPTIMIZE C            BUILD C                       |
  |            USERFACE C                                          |
  |                                                                |
   ================================================================
*/


/*
   ================================================================
  |                          RTRACE C MODULE                       |
   ================================================================
*/

void
        transformvector (double (*m)[4],double *vec,double *invvec),
        transformpoint  (double (*m)[4],double *pt,double *invpt),
        raycast         (struct OBJCT *scene,int x,int y,char *c),
        testray         (struct OBJCT *o,double *pt,double *dirn,
                         double *nrml,double *dst),
        generateray     (int x,int y,double *p,double *d),
        preprocess      (struct OBJCT * *scene),
        raytrace        (struct OBJCT * *scene),
        rgbtrace        (struct OBJCT *scene),
        normalize       (double *v)

int
        linecompress    (unsigned char *bufin,unsigned char *bufout,
                         int y,int x1, int x2),
        inshadow        (struct OBJCT *scene,struct OBJCT *obj,
                         double *pt,double *dirn,double tl),


struct OBJCT *
        nextobject      (struct OBJCT *o,struct nde * *n),


/*
   ================================================================
  |                       RAYINTER C  MODULE                       |
   ================================================================
*/

void
        tracecube       (double *pt,double *dirn,double *nrm,
                         double *nearest),
        tracesphere     (double *pt,double *dirn,double *nrm,
                         double *nearest),
        tracecylinder   (double *pt,double *dirn,double *nrm,
                         double *nearest),
        tracecone       (double *pt,double *dirn,double *nrm,
                         double *nearest),

int
        stracecube      (double *pt,double *dirn,double tl),
        stracesphere    (double *pt,double *dirn double tl),
        stracecylinder  (double *pt,double *dirn,double tl),
        stracecone      (double *pt,double *dirn,double tl)
```

```
/*
   ================================================================
  |                      OPTIMIZE C  MODULE                        |
   ================================================================
*/

void
        transformvolume (double (*m)[4],double (*v)[3],double (*r)[3]),
        calczdepth      (double (*v)[3],struct OBJCT *o),
        calcextent      (double (*v)[3],struct OBJCT *o),
        makegrid        (struct OBJCT *scene),
        projectvolume   (double (*v)[3]),

int
        compare         (struct OBJCT *a,struct OBJCT *b),

struct nde *
        getnode         (int i,int j),


struct OBJCT *
        sortlst        (struct OBJCT *p,int (cdecl *compare)()),

/*
   ================================================================
  |                        SHADE C  MODULE                         |
   ================================================================
*/

void
        pgashade       (unsigned char *c,struct OBJCT *obj,double *ray,
                        double *lght,double *nrml,int x,int y,int shad),
        rgbshade       (unsigned char *c,struct OBJCT *obj,double *ray,
                        double *lght,double *nrml,int x,int y,int shad),

/*
   ================================================================
  |                      PGADPEND C  MODULE                        |
   ================================================================
*/

void
        pgatrace        (struct OBJCT *scene),
        generatergbup   (int c,int r,int g,int b),
        generatergbdown (int c,int r,int g,int b),
        rendervolume    (double (*v)[3],int c),
        loadpgafile     (char *str),
        initpga         (void),
        quitpga         (void),

int
        readcolors      (char *str),
        savecolors      (char *str),

/*
   ================================================================
  |                        BUILD C  MODULE                         |
   ================================================================
*/

void
        matscale        (double (*m)[4] double sx double sy double sz)
        mattranslate    (double (*m)[4],double tx,double ty,double tz),
        ptscale         (double *p,double sx,double sy,double sz),
        pttranslate     (double *p,double tx,double ty,double tz),
        matrotx         (double (*m)[4],double deg)
        matroty         (double (*m)[4],double deg),
```

```
        matrotz         (double (*m)[4],double deg),
        ptrotx          (double *p,double a),
        ptroty          (double *p,double a),
        ptrotz          (double *p,double a),

int
        inversematrix   (double (*c)[4],double (*b)[4]),


struct OBJCT *
        getobject       (void),


/*
   ==========================================================
  |                    USERFACE C  MODULE                    |
   ==========================================================
*/

int
        setwindow               (double x1,double x2,double y1,
                                 double y2),
        setviewport             (int x1,int x2,int y1,int y2),
        setlightsource          (double x,double y,double z),
        setprojectiondirection  (double x,double y,double z),
        setlighttype            (enum lighttype lght),
        setprojection           (enum projtype proj),
        setformat               (enum format frm),
        setviewplanedistance    (double dist),
        setambientlight         (double amb),
        setfileoutput           (char *str),
        setcompression          (int cmp),
        setscreen               (int scr),
        setbackgroundcolor      (int col),
        setfillcolor            (int col),
        setshadows              (int shd),
        setdither               (int dth),
        setpixelbuffer          (int buf),
        setsortlist             (int srt),
        setextents              (int extn),
        setgrid                 (int grd),

int
        readviewport            (int *x1,int *x2,int *y1,int *y2),
        readprojectiondirection (double *x,double *y,double *z),
        readlightsource         (double *x,double *y,double *z),
        readwindow              (double *x1,double *x2,
                                 double *y1,double *y2),
        readlighttype           (enum lighttype *lght),
        readprojection          (enum projtype *proj),
        readfileoutput          (struct _iobuf *fp),
        readformat              (enum format *frm),
        readviewplanedistance   (double *dist),
        readambientlight        (double *amb),
        readcompression         (int * cmp),
        readextents             (int *extn),
        readscreen              (int *scr),
        readbackgroundcolor     (int *col),
        readfillcolor           (int *col),
        readshadows             (int *shd),
        readdither              (int *dth),
        readpixelbuffer         (int *buf),
        readsortlist            (int *srt),
        readgrid                (int *grd),
```

```
/*
   ================================================================
  | This file contains the "typedef" declarations that define      |
  | the various enumerative and structure types                    |
   ================================================================
*/


#define    INFINITY    100000000 0
#define    GRIDROW     20
#define    GRIDCOL     20
#define    ON   1
#define    OFF  0


/* TYPEDEF DECLARATIONS
   ================================================================
  | The following global types are defined  -                      |
  |                                                                |
  | lightytype --> enumerative type for defining point or          |
  |                directional light source                        |
  |                                                                |
  | projtype   --> enumerative type for defining parallel or       |
  |                perspective  projection                         |
  |                                                                |
  | format     --> enumertive type for defining ray tracing mode   |
  |                i e  red green blue (rgb) or Professional       |
  |                Graphics  Adaptor (pga) mode                    |
  |                                                                |
  |                                                                |
  |  OBJECTTYPE    enumerative type for primitive solid names      |
  |                                                                |
  |  VECTOR        1x3 double array for vector X,Y,Z coordinates   |
  |                                                                |
  |  POINT         1x3 double array for point X,Y,Z coordinates    |
  |                                                                |
  |  MATRIX        4x4 double array for matrix transforms          |
  |                                                                |
  |  NODE          structure type used for forming linked list     |
  |                of pointers to objects (see grid optimization)  |
  |                                                                |
  |                next       -->  pointer to next node           |
  |                ptr        -->  pointer to primitive           |
  |                                                                |
  |  OBJECT        structure type for primitive definition  -      |
  |                                                                |
  | fields         next       -->  pointer to next primitive      |
  |                primitive  -->  type of primitive              |
  |                transform  -->  transformation matrix          |
  |                inverse    -->  inverse transformation matrix  |
  |                clr        -->  primitive color (0  15)        |
  |                xmin, xmax ==>  screen rectangle enclosing     |
  |                ymin, ymax      projected bounding volume      |
  |                znear,zfar -->  nearest & furthest Z coords of |
  |                                transformed bounding volume,   |
  |                                w r t  primary ray origin      |
  |                ka, kd, ks -->  ambient, diffuse and specular  |
  |                                reflection constants (should   |
  |                                add up to one)                 |
  |                pwr        -->  specular power constant        |
  |                rnd        -->  roughness (0  2)               |
  |                cmy        -->  cyan, magenta & yellow         |
  |                                reflection ratios              |
   ================================================================
*/
```

```
enum  lighttype    { lightdirection, lightpoint },
enum  projtype     { parallel, perspective },
enum  format       { rgb, pga },


typedef        enum  {cube,sphere,cylinder,cone}   OBJECTTYPE,

typedef        double    VECTOR[3],
typedef        double    POINT[3],
typedef        double    MATRIX[4][4],

typedef        struct OBJCT  {
                             struct OBJCT * next,
                             OBJECTTYPE     primitive,
                             MATRIX         transform,inverse,
                             int            clr,
                                            xmin,xmax,ymin,ymax,
                             double         znear, zfar,
                                            ka,kd,ks,pwr,
                                            rnd,
                                            cmy[3],
                             } OBJECT,

typedef        struct nde    {
                             struct nde * next,
                             OBJECT * ptr,

                             } NODE,
```

```
/*
   ================================================================
  | This file contains declarations of the various global flags,   |
  | parameters and variables used bty the raytracer                |
   ================================================================
*/

#include   "typedef h"


/*
   ================================================================
  |                GLOBAL PARAMETER DECLARATIONS                   |
   ================================================================
*/

extern int
                fillcolor,              /* fill color              */
                background,             /* background color        */
                vx1,vx2,vy1,vy2,        /* viewport                */

extern double
                ambient,                /* ambient light intensity */
                wx1, wx2, wy1, wy2,     /* window on viewplane     */
                viewplanedist,          /* viewplane distance from */

extern POINT
                viewpoint,              /* perspective viewpoint   */
                light,                  /* light source            */


extern VECTOR
                projection,             /* projection direction    */

extern FILE  *
                outfile,                /* output file for image   */


/*
   ================================================================
  |                         GLOBAL FLAGS                           |
   ================================================================
*/

extern int
                PERSPECTIVE,            /* => parallel projection  */
                SHADOWS,                /* No shadows              */
                DITHER,                 /* No dither on pixels     */
                FILEOUT,                /* No output file generated*/
                PGA,                    /* Generate PGA image      */
                COMPRESS,               /* Compress PGA file format*/
                SCREEN,                 /* Output image to screen  */
                POINTSOURCE,            /* light interpreted as a  */
                                        /* vector and not a point  */

                EXTENTS,                /* Use extents             */
                SORTLIST,               /* Sort scene list         */
                PIXELBUFFER,            /* Use pixel buffer        */
                GRID,                   /* Use grid                */


/*
   ================================================================
  |                         GLOBAL VARIABLES                       |
   ================================================================
*/
```

```
extern char *
                nameof[4],              /* string names of primitives */

extern int
                dither4[4][4],          /* 4x4 dither matrix          */

extern double
                xfacvw, yfacvw,         /* viewport -> window X-ratio */
                xfacwv, yfacwv,         /* window -> viewport Y-ratio */

extern NODE *
                grid[GRIDROW][GRIDCOL], /* pointers to linked lists*/

extern POINT
                volume[4][8],           /* 8 vertex bounding volume   */
                                        /* for each primitive type    */
```

```
/* TRANSFORMATION MATRIX FUNCTIONS MODULE
   ======================================================================
  | This  module  contains  functions  for  modifying  a  4x4 |
  | transformation matrix to incorporate a translation, rotation or |
  | scaling operation, in addition to functions for translating, |
  | rotating or scaling an individual point                  |
  |                                                          |
  | Functions -                                              |
  |               inversematrix          getobject           |
  |               mattranslate           pttranslate         |
  |               matscale               ptscale             |
  |               matrotx                ptrotx              |
  |               matroty                ptroty              |
  |               matrotz                ptrotz              |
   ======================================================================
*/

#include   <stdio h>
#include   <malloc h>
#include   <math h>
#include   "typedef h"   /* structure and other typedef definitions*/
#include   "function h"  /* function prototype declarations       */


/* MATRIX FUNCTIONS
   ======================================================================
  | The  following functions are used to build up a transform matrix |
  | for a primitive, comprising of translation, rotation and scaling |
  | operations which will transform it from its own local coordinate |
  | system into the world coordinate system, with a different |
  | position, size and orientation                           |
  |                                                          |
  | matrotx          alter matrix to take in rotation about X axis |
  | matroty           "     "    "     "      "       "    Y axis |
  | matrotz           "     "    "     "      "       "    Z axis |
  |                                                          |
  | matscale         alter matrix to take in scaling along X,Y,Z axes |
  |                                                          |
  | mattranslate     alter matrix to take in translation along X,Y,Z |
  |                                                          |
  | inversematrix    Calculate inverse of a 4x4 matrix       |
  |                                                          |
  | NOTE             Since rotations are not commutative ie the |
  |                  order  in  which  they are carried out is |
  |                  significant, there is a seperate function for |
  |                  each axis                               |
   ======================================================================
*/


void matrotx (m,deg)
MATRIX m,                          /* transform matrix  */
double deg,                        /* degrees to rotate */

/*
   ======================================================================
  | Modifies specified transform matrix, m, to take in a rotation by |
  | deg degrees about the X axis (result returned in m)      |
   ======================================================================
*/

{double c,s,rad,t[4],
 int i,

  for (i=0, i<4, i++)              /* used to help optamize the  */
   t[i] = m[i][1],                 /* matrix multiplication      */
```

```
  rad = deg/57 295779,          /* convert degrees to radians  */
  c = cos(rad),
  s = sin(rad),

  for (i=0, i<4, i++)
   { m[i][1] = t[i]*c - m[i][2]*s,    /* modify matrix  to take in  */
     m[i][2] = m[i][2]*c + t[i]*s,    /* the rotation  Optimize the */
    }                                 /* multiplication by omitting */
                                      /* columns of zeros           */
}


void matroty (m,deg)
MATRIX m,
double deg,
/*
   ================================================================
  | Modifies specified transform matrix, m, to take in a rotation by |
  | deg degrees about the Y axis (result returned in m)              |
   ================================================================
*/

{double c,s,rad,t[4],
int i,

  for (i=0, i<4, i++)
   t[i] = m[i][0],

  rad = deg/57 295779,
  c = cos(rad),
  s = sin(rad),
  for (i=0, i<4, i++)
   { m[i][0] = t[i]*c + m[i][2]*s,
     m[i][2] = m[i][2]*c - t[i]*s,
    }
}


void matrotz (m,deg)                  /* similar to matrotx function above */
MATRIX m,
double deg,
/*
   ================================================================
  | Modifies specified transform matrix, m, to take in a rotation by |
  | deg degrees about the Z axis (result returned in m)              |
   ================================================================
*/

{double c,s,rad,t[4],
int i,

  for (i=0, i<4, i++)
   t[i] = m[i][0],

  rad = deg/57 295779,
  c = cos(rad),
  s = sin(rad),
  for (i=0, i<4, i++)
   { m[i][0] = t[i]*c - m[i][1]*s,
     m[i][1] = m[i][1]*c + t[i]*s,
    }
}


void matscale(m,sx,sy,sz)
MATRIX m,                         /* transformation matrix          */
double sx,sy sz                   /* scaling values for X  Y Z axes */
```

```
/*
   ==========================================================
  | Modifies specified transform matrix, m, to take in scaling |
  | (w r t  the origin) by factors of sx, sy and sz along the X, Y, |
  | and Z axes respectively (result returned in m)             |
   ==========================================================
*/

{int i,

  for (i=0, i<4, i++)
   { m[i][0] *= sx,              /* modify matrix to take in       */
     m[i][1] *= sy,              /* scaling operations             */
     m[i][2] *= sz,              /* Order of performing the        */
    }                            /* scaling operations unimportant */
}


void mattranslate(m,tx,ty,tz)
MATRIX m,                  /* transformation matrix              */
double tx,ty,tz,           /* translation values for X, Y Z axes  */

/*
   ==========================================================
  | Modifies specified transform matrix, m, to take in translation |
  | (w r t  the origin) of tx, ty and tz units along the X, Y and Z |
  | axes respectively (result returned in m)                   |
   ==========================================================
*/

{int i,

 for (i=0, i<4, i++)
    {m[i][0] += m[i][3]*tx,         /* modify matrix to take in    */
     m[i][1] += m[i][3]*ty,         /* translation operations      */
     m[i][2] += m[i][3]*tz,         /* Order of carrying out the   */
     }                              /* translations doesn't matter */
}



int inversematrix(c,b)
MATRIX c,b,

/*
   ==========================================================
  | Returns in b,  the inverse of the 4x4 transform matrix specified |
  | in c (which remains unchanged)  The inverse is calculated by |
  | performing on an identity matrix, the same elementary row |
  | operations required to reduce matrix c to the identity matrix  |
   ==========================================================
*/

{double d,
 MATRIX a,
 int i,j,row,found,
 double tmp,

 for (i=0, i<4, i++)        /* don't want to alter matrix c  */
   for (j=0, j<4, j++)      /* so copy to local matrix, a    */
    a[i][j] = c[i][j],



 for (i=0, i<4, i++)                /* initialize b to identity    */
  for (j=0, j<4, j++)               /* matrix                      */
    if (i == j) b[i][j] = 1 0,
      else b[i][j] = 0 0
```

```
for (row=0, row<4, row++)
 {if (a[row][row] != 1 0)          /* diagonal entry not a 1 ?    */
    { if (a[row][row] != 0 0)      /* if not then, if entry not a */
                                   /* zero, multiply row by its   */
       {d = 1 0/a[row][row],       /* reciprocal to make a one    */
        for (i=0, i<4, i++)
         { a[row][i] *= d,
           b[row][i] *= d,
           }
       }
       else                                /* if entry a zero, then */
         {                                 /* search in column below*/
          found = 0,  i = row+1,           /* for non-zero entry    */
          while ((!found) && (i < 4))
          {
           if (a[i][row] != 0)             /* found one ?           */
            {found = 1,                    /* Then take reciprocal, */
             d = 1 0/a[i][row],            /* d, and add d times    */
                                           /* this row to the one   */
             for (j=0, j<4, j++)           /* above, to get a one in*/
              {a[row][j] += d * a[i][j],   /* the required diagonal*/
               b[row][j] += d * b[i][j],
                }
              }
               i += 1,
             }
           if (!found) return  0,          /* no non-zero entry in  */
      }                                    /* column below => matrix */
     }                                     /* not invertible        */
    for (i=row+1, i<4, i++)
      {
       if (a[i][row] != 0 0)               /* have a one in diagonal */
      -   { d = -a[i][row],                /* so set all entries in  */
            for (j=0, j<4, j++)            /* col below to zero      */
              { a[i][j] += d * a[row][j],
                b[i][j] += d * b[row][j],
                }
             }
        }

    for (i=row-1, i>=0, i--)
      {
        if (a[i][row] != 0 0)              /* now set all entries   */
          { d = -a[i][row],                /* in col above to zero  */
            for (j=0, j<4, j++)
              { a[i][j] += d * a[row][j],
                b[i][j] += d * b[row][j],
                }
             }
        }
   }

   return 1,                               /* indicate success */
}

/* POINT MANIPULATION FUNCTIONS
   ============================================================
  | The following functions can be  used  to  rotate,  translate  or |
  | scale an individual point with respect to the origin             |
  |                                                                  |
  | ptrotx             rotate point about X axis                     |
  | ptroty                 '           '   Y axis                    |
  | ptrotz                 '    '      "   Z axis                    |
  | ptscale            scale point w r t  origin                     |
  | pttranslate        translate point w r t  origin                 |
   ============================================================
*/
```

```
void ptrotx (p,a)
POINT p,
double a,

{double c,s,rad,t[3],
int i,

  for (i=0, i<3, i++)
   t[i] = p[i],

  rad = (double)a/57 2975,
  c = cos(rad),
  s = sin(rad),

    p[1] = t[1]*c - t[2]*s,
    p[2] = t[2]*c + t[1]*s,
}


void ptroty (p,a)
POINT p,
double a,

{double c,s,rad,t[3],
int i,

  for (i=0, i<3, i++)
   t[i] = p[i],

  rad = (double)a/57 2975,
  c = cos(rad),
  s = sin(rad),
    p[0] = t[0]*c + t[2]*s,
    p[2] = t[2]*c - t[0]*s,
}


void ptrotz (p,a)
POINT p,
double a,

{double c,s,rad,t[3],
int i,

  for (i=0, i<3, i++)
   t[i] = p[i],

  rad = (double)a/57 2975,
  c = cos(rad),
  s = sin(rad),
    p[0] = t[0]*c - t[1]*s,
    p[1] = t[1]*c + t[0]*s,
}


void ptscale(p,sx,sy,sz)
POINT p,
double sx,sy,sz,

{
  p[0] *= sx,   p[1] *= sy,   p[2] *= sz,
}
```

```
void pttranslate(p,tx,ty,tz)
POINT p,
double tx,ty,tz,

{
  p[0] += tx,
  p[1] += ty,
  p[2] += tz,
}


/* OBJECT FUNCTIONS
   ================================================================
  | zzgetobjectzzz zzzzzuses malloc to get space for new OBJECT |
  |                    structure and return pointer to it       |
   ================================================================
*/

OBJECT *getobject()
/*
   ================================================================
  | Returns a pointer to space allocated for new OBJECT structure  |
  | The transformation matrix is initialized to the identity matrix |
  | and the next pointer field to NULL                             |
   ================================================================
*/
{int i,j,
OBJECT *p,

 p = (OBJECT *)malloc(sizeof(OBJECT)),
   for (i=0, i<4, i++)
    for (j=0, j<4, j++)
      if (i == j) p->transform[i][j] = 1 0,
        else p->transform[i][j] = 0 0,
  p->next = NULL,
  p->primitive = sphere,
  p->ka   = 1 0,
  p->kd   = 0 4,
  p->ks   = 0 6,
  p->rnd  = 0 8,
  p->pwr  = 20,
  p->clr  = 0,
  p->cmy[0] = p->cmy[1] = p->cmy[2] = 0 3,
 return p,
}
```

```
/* USER INTERFACE MODULE
   ================================================================
   | The functions contained in this module provide the user with a |
   | means to read and change the various viewing parameters,       |
   | optimizations and output options of the ray tracer             |
   |                                                                |
   |  Parameter Setting Functions       Flag Setting Functions      |
   |                                                                |
   |    setprojectiondirection              setprojection           |
   |    setviewplanedistance                setlighttype            |
   |    setbackgroundcolor                  setshadows              |
   |    setambientlight                     setdither               |
   |    setlightsource                      setformat               |
   |    setfileoutput                       setscreen               |
   |    setfillcolor                        setcompression          |
   |    setviewport                         setpixelbuffer          |
   |    setwindow                           setextents              |
   |                                        setsortlist             |
   |                                        setgrid                 |
   |                                                                |
   |  Parameter Reading Functions       Flag Reading Functions      |
   |                                                                |
   |    readprojectiondirection             readprojection          |
   |    readviewplanedistance               readlighttype           |
   |    readbackgroundcolor                 readshadows             |
   |    readambientlight                    readdither              |
   |    readlightsource                     readformat              |
   |    readfileoutput                      readscreen              |
   |    readfillcolor                       readpixelbuffer         |
   |    readviewport                        readsortlist            |
   |    readwindow                          readextents             |
   |                                        readgrid                |
   |  * Other functions  -                                          |
   |                        generatergbdown                         |
   |                        loadpgafilecom                          |
   |                        generatergbup                           |
   |                        loadpgafile                             |
   |                        readcolors                              |
   |                        savecolors                              |
   |                        initpga                                 |
   |                        quitpga                                 |
   |                                                                |
   |  * All of these functions interact directly with the PGA card  |
   |    and are contained in the PGADPEND C module                  |
   ================================================================
*/

#include    <stdio h>
#include    "global h"     /* global parameters flags & variables */
#include    "function h"   /* function prototype declarations     */

/* PARAMETER SETTING FUNCTIONS
   ================================================================
   | The following functions each set one or more of the raytracer  |
   | viewing and lighting parameters  -                             |
   |                                                                |
   |       Function                        Parameters Set           |
   |                                                                |
   | setprojectiondirection       parallel projection direction     |
   | setviewplanedistance         view plane distance from origin   |
   | setbackgroundcolor           background color                  |
   | setambientlight              ambient light intensity           |
   | setlightsource               light source position or direction|
   | setfileoutput                output image to file              |
   | setfillcolor                 fillcolor for bounding volumes    |
   | setviewport                  viewport                          |
   | setwindow                    window on viewplane               |
   ================================================================ */
```

```
int setprojectiondirection(x,y,z)
double x,y,z,
/*
   ============================================================
  | Sets the direction for parallel projection to the specified XYZ |
  | direction through the global variable projection The default is |
  | the direction (0,0,-1)  ie orthographic projection              |
   ============================================================
*/
{
  projection[0] = x,
  projection[1] = y,
  projection[2] = z,
}


int setviewplanedistance(dist)
double dist,
/*
   ============================================================
  | Default viewplane is the Z=0 plane  This function changes it to |
  | Z=dist plane by setting the global variable viewplanedist       |
  ============================================================
*/
{
  viewplanedist = dist,
}

int setbackgroundcolor(col)
int col,
/*
   ============================================================
  | The global variable background is the color to which all pixels |
  | that do not intersect any object are set  (default=113)         |
   ============================================================
*/
{
  background = col,
}


int setambientlight(amb)
double amb,
/*
   ============================================================
  | Set ambient light intensity through global variable ambient  |
  | Default = 0 2                                                |
   ============================================================
*/
{
  ambient = amb,
}


int setlightsource(x,y,z)
double x,y,z,
/*
   ============================================================
  | Sets the elements of the global variable light, which can be |
  | interpreted as a point or a vector (see setlighttype below)  |
  | Default is (0,0,1)                                           |
   ============================================================
*/
{
  light[0] = x,
  light[1] = y,
  light[2] = z,
}
```

```
int setfileoutput(str)
char *str,
/*
   ==========================================================
  | Specifies  that the raytraced image be sent to a file whose name |
  | is specified in str  (default is no file output)                 |
   ==========================================================
*/
{
  outfile = fopen(str,"wb"),
  FILEOUT = ON,
}


int setfillcolor(col)
int col,
/*
   ==========================================================
  | Fillcolor is the color in which the transformed bounding volumes |
  | of  the  primitives  are  filled  when  using  the  pixelbuffer  |
  | optimization  It  can  be  any  color other than the background  |
  | color  (Default is color 112)                                    |
   ==========================================================
*/
{
  fillcolor = col,
}


int setviewport(x1,x2,y1,y2)
int x1,x2,y1,y2,
/*
   ==========================================================
  | Screen viewport defined by the global  variables  (vx1,vy1)  and |
  | (vx2,vy2) the lower left and upper right coordinates  Default is |
  | (0,0) and (639,479), PGA maximum resolution                      |
   ==========================================================
*/
{
  vx1 = x1,  vx2 = x2,
  vy1 = y1,  vy2 = y2,
}


int setwindow(x1,x2,y1,y2)
double x1,y1,x2,y2,
/*
   ==========================================================
  | window  on  viewing  plane  defined  by  the  global  variables |
  | (wx1,wy1)  and  (wx2,wy2)  the  lower  left  and  upper  right   |
  | coordinates respectively  Default is (-100 100) & (-100,100)     |
   ==========================================================
*/
{
  wx1 = x1,  wx2 = x2,
  wy1 = y1,  wy2 = y2,
}
```

```
/* FLAG SETTING FUNCTIONS
   ==========================================================
   | The following functions each set one of the raytracers global |
   | flags that control optimization, shadows etc                  |
   |                                                               |
   |     Function                    Flag affected                 |
   |                                                               |
   |  setcompression        perspective or parallel projection     |
   |  setprojection         perspective or parallel projection     |
   |  setlighttype          point or directional light             |
   |  setshadows            include/exclude shadows                |
   |  setdither             do/don't apply dither matrix to pixels |
   |  setformat             select PGA or RGB format               |
   |  setscreen             display generated image on screen      |
   |                                                               |
   |  setpixelbuffer        pixel buffer optimization on/off       |
   |  setsortlist           do/don't sort primitive list           |
   |  setextents            extents optimization on/off            |
   |  setgrid               rectangular grid optimization on/off   |
   ==========================================================
*/


int setcompression(cmp)
int cmp,
/*
   ==========================================================
   | If the COMPRESS flag is set and the output file is set to PGA |
   | format, a compressed image file is generated by run length   |
   | encoding the original image  A description of the run encoding |
   | method used can be found in the function runencode in the    |
   | raytracing module RTRACE C                                   |
   ==========================================================
*/
{
  COMPRESS = cmp,
}

int setprojection(proj)
enum projtype  proj,
/*
   ==========================================================
   | Defines a perspective or parallel projection by appropiately |
   | setting the global variable PERSPECTIVE to ON (1) or OFF (0) |
   | The default is OFF ie parallel projection                    |
   ==========================================================
*/
{
   if (proj == perspective)
     PERSPECTIVE = ON,
   else
     PERSPECTIVE = OFF,
}


int setlighttype(lght)
enum lighttype  lght,
/*
   ==========================================================
   | The XYZ elements of the global variable light can be interpreted |
   | coordinates of a light source at the point (X,Y,Z), or as a  |
   | vector in the direction of a light source at infinity  The   |
   | latter is slightly less expensive, computationally, to model |
   | since all light rays are then parallel  Default is direction |
   | interpretion ie  POINTSOURCE=OFF                             |
   ==========================================================
*/
```

```
{
  if (light == lightpoint)
    POINTSOURCE = ON,
   else POINTSOURCE = OFF,
}

int setshadows(shd)
int shd,
/*
   ============================================================
  | If the global variable SHADOWS is set, shadows will be    |
  | incorporated into the raytraced image  (This greatly increases |
  | the time taken to render an image and so is OFF by default)  |
   ============================================================
*/
{
    SHADOWS = shd,
}

int setdither(dth)
int dth,
/*
   ============================================================
  | If the global variable DITHER is set, a a 4x4 dither matrix is |
  | applied to calculated pixel intensities  (Default is OFF)    |
   ============================================================
*/
{
    DITHER = dth,
}


int setformat(frm)
enum format frm,
/*
   ============================================================
  | Selects either PGA or RGB format for image  In PGA format, the |
  | image is displayed on the Professional Graphichs Display  The |
  | RGB  mode does not generate a screen image but is used to affect |
  | the output file format  -                                  |
  |                                                            |
  | Output file format can be either rgb or pga (default is pga)  |
  |                                                            |
  | pga  zzzzEach pixel stored as one byte representing a  color  in |
  |          the  range  0  255  which is used by the pga card as an |
  |          index into a color  table  containing  a  12-bit  entry |
  |          which  determines  the  color actually displayed on the |
  |          screen                                            |
  |                                                            |
  | rgb  zzzzEach pixel stored as three bytes,  one for each of  its |
  |          red, green and blue intensities                   |
   ============================================================
*/
{
  if (frm == pga)
   PGA = ON,
  else
   PGA = OFF,
}

int setscreen(scr)
int scr,
/*
   ============================================================
  | If  the  global variable SCREEN is set, the raytraced image will |
  | be displayed  line by line  on screen as it is generated   |
  | Default is ON                                              |
   ============================================================ */
```

```
{
  SCREEN = scr,
}


int setpixelbuffer(buf)
int buf,
/*
   ================================================================
  | If the global variable PIXELBUFFER is set, primitive bounding |
  | volumes are rendered rendered on screen, filled in fillcolor |
  | (another global variable), to reduce rendering time          |
  | Default is ON                                                |
   ================================================================
*/
{
    PIXELBUFFER = buf,
}

int setsortlist(srt)
int srt,
/*
   ================================================================
  | If the global variable SORTING is set, the list of primitives |
  | passed to the raytracer is sorted in order of increasing |
  | distance from the ray to reduce rendering time               |
  | Default is ON                                                |
   ================================================================
*/
{
    SORTLIST = srt,
}


int setextents(extn)
int extn,
/*
   ================================================================
  | If the global variable EXTENTS is set (=1), screen extents are |
  | generated for each primitive to reduce rendering time        |
  | Default is ON                                                |
   ================================================================
*/
{
    EXTENTS = extn,
}


int setgrid(grd)
int grd,
/*
   ================================================================
  | If GRID, a global variable, is set, the screen is partitioned |
  | into a set number of rectangles each of which has associated |
  | with it a set of pointers to primitives in the scene list whose |
  | screen enclosures cross the rectangle                        |
  | Default is ON                                                |
   ================================================================
*/
{
    GRID = grd,
}
```

```
/* PARAMETER READING FUNCTIONS
   ==========================================================
   | The following functions are used to read the current  values  of |
   | the viewing and lighting parameters of the raytracer  -          |
   |                                                                  |
   |       Function                      Parameters Returned          |
   |                                                                  |
   |  readprojectiondirection       parallel projection direction     |
   |  readviewplanedistance         view plane distance from origin   |
   |  readbackgroundcolor           background color                  |
   |  readambientlight              ambient light intensity           |
   |  readlightsource               light source position/direction   |
   |  readfileoutput                output file for image             |
   |  readfillcolor                 fillcolor for bounding volumes    |
   |  readviewport                  viewport                          |
   |  readwindow                    window on viewplane               |
   ==========================================================
*/

int readprojectiondirection(x,y,z)
double *x,*y,*z,
{
  *x = projection[0],
  *y = projection[1],
  *z = projection[2],
}

int readviewplanedistance(dist)
double *dist,
{
   *dist = viewplanedist,
}


int readbackgroundcolor(col)
int *col,
{
  *col = background,
}


int readambientlight(amb)
double *amb,
{
  *amb = ambient,
}


int readlightsource(x,y,z)
double *x,*y,*z,
{
  *x = light[0],
  *y = light[1],
  *z = light[2],
}

int readfileoutput(fp)
FILE *fp,
{
  fp = outfile,
}


int readfillcolor(col)
int *col,
{
  *col = fillcolor,
}
```

```
int readviewport(x1,x2,y1,y2)
int *x1,*x2,*y1,*y2,
{
  *x1 = vx1, *x2 = vx2,
  *y1 = vy1, *y2 = vy2,
}


int readwindow(x1,x2,y1,y2)
double *x1,*y1,*x2,*y2,
{
  *x1 = wx1,  *x2 = wx2,
  *y1 = wy1,  *y2 = wy2,
}



/* FLAG READING FUNCTIONS
   ==========================================================
  |                                                          |
  | The following functions return the current settings for the |
  | various global flags used by the raytracer -             |
  |                                                          |
  |     Function                    Flag Returned            |
  |                                                          |
  |  readcompression        perspective or parallel projection |
  |  readprojection         perspective or parallel projection |
  |  readformat             PGA or RGB image format          |
  |  readscreen             screen image on/off              |
  |  readlighttype          point or directional light       |
  |  readshadows            include/exclude shadows          |
  |  readdither             do/don't apply dither matrix to pixels |
  |                                                          |
  |  readpixelbuffer        pixel buffer optimization  on/off |
  |  readsortlist           do/don't sort primitive list     |
  |  readextents            extents optimization on/off      |
  |  readgrid             rectangular grid optimization on/off |
  |                                                          |
   ==========================================================
*/


int readcompression(cmp)
int *cmp,
{
  *cmp = COMPRESS,
}


int readprojection(proj)
enum projtype  *proj,
{
   if (PERSPECTIVE == ON)
     *proj = perspective,
   else
     *proj = parallel,
}


int readlighttype(lght)
enum lighttype  *lght,
{
  if (POINTSOURCE == ON)
    *lght = lightpoint,
   else
    *lght = lightdirection,
}
```

```
int readshadows(shd)
int *shd,
{
    *shd = SHADOWS,
}


int readdither(dth)
int *dth,
{
    *dth = DITHER,
}

int readformat(scr)
enum format *scr,
{
  if (PGA == ON)
   *scr = pga,
  else
   *scr = rgb,
}


int readscreen(scr)
int *scr,
{
  *scr = SCREEN,
}

int readpixelbuffer(buf)
int *buf,
{
    *buf = PIXELBUFFER,
}


int readsortlist(srt)
int *srt,
{
    *srt = SORTLIST,
}



int readextents(extn)
int *extn,
{
    *extn = EXTENTS,
}


int readgrid(grd)
int *grd,
{
    *grd = GRID,
}
```

```
/* RAYTRACING MODULE    RTRACE C
   ============================================================
   | This module contains the raytracing functions themselves, namely |
   | raytrace and raycast, which coordinate the overall raytracing |
   | operation                                                  |
   |                                                            |
   |    Raytracing Functions          Utility Functions         |
   |                                                            |
   |        raytrace                   nextobject               |
   |      * pgatrace                   transformvector          |
   |        rgbtrace                   transformpoint           |
   |        inshadow                   generateray              |
   |        raycast                    preprocess               |
   |        testray                    normalize                |
   |                                                            |
   |  * function uses PGA library function calls, code so       |
   |    contained in PGA dependent module PGADPEND C            |
   ============================================================
*/


#include   <stdio h>
#include   <malloc h>
#include   <math h>
#include   "typedef h"   /* typedef definitions for POINT etc   */
#include   "function h"  /* function prototype declarations     */


/* DEFAULT PARAMETER VALUES
   ============================================================
   | Define default values for various global parameters such as |
   | window, viewport etc  Values defined below can be accessed by |
   | the user through the appropiate function calls described in the |
   | user interface module USERFACE C                           |
   ============================================================
*/

int
          fillcolor    = 112,          /* fill color              */
          background   = 113,          /* background color        */
          vx1 = 0, vx2 = 639,          /* viewport                */
          vy1 = 0, vy2 = 479,


double
          ambient = 0 2,               /* ambient light intensity */
          viewplanedist     = 500 0,   /* viewplane distance from */
                                       /* origin (along Z axis)   */
          wx1 = -100 0, wx2 = 100 0,   /* window on viewplane     */
          wy1 = -100 0, wy2 = 100 0,


POINT
          viewpoint  = { 0, 0, 0 },    /* perspective viewpoint   */
          light      = { 1, 1, 1 },    /* light source            */


VECTOR
          projection = { 0, 0, -1 },   /* projection direction    */


FILE  *
          outfile    = NULL,           /* output file for image   */
```

```
/* GLOBAL FLAGS
   ============================================================
   |                                                          |
   | Define default values for various global flags that define |
   | whether various options such as shadows, optimizations etc  are |
   | defined below  Each flag can be either ON or OFF  The default |
   | values defined below can be accessed and set ON or OFF by the |
   | user through the appropiate function calls described in the user |
   | interface module USERFACE C                              |
   |                                                          |
   ============================================================
*/

 int
          PERSPECTIVE     =   OFF,     /* => parallel projection  */
          POINTSOURCE     =   OFF,     /* light interpreted as a  */
                                       /* vector and not a point  */
          SHADOWS         =   OFF,     /* No shadows              */
          DITHER          =   OFF,     /* No dither on pixels     */
          FILEOUT         =   OFF,     /* No output file generated*/
          COMPRESS        =    ON,     /* Compress PGA file format*/
          SCREEN          =    ON,     /* Display screen image    */
          PGA             =    ON,     /* Generate PGA format     */

/*optimizations */

          EXTENTS         =   ON,      /* Use extents             */
          SORTLIST        =  OFF,      /* Sort scene list         */
          PIXELBUFFER     =   ON,      /* Use pixel buffer        */
          GRID            =   ON,      /* Use grid                */
```

```
/* GLOBAL VARIABLES
   ============================================================
   |                                                          |
   | The  Following  global variables and are used exclusively by the |
   | raytracer                                                |
   |                                                          |
   | nameof  -->   array of pointers to string names of primitives |
   |                                                          |
   | dither4 -->   4x4 dither matrix                          |
   |                                                          |
   | xfacvw  -->   viewport to window X ratio                 |
   | yfacvw  -->   viewport to window Y ratio                 |
   |                                                          |
   | xfacwv  -->   window to viewport X ratio                 |
   | yfacwv  -->   window to viewport Y ratio                 |
   |                                                          |
   | grid    -->   GRIDROW x GRIDCOL 2D array of pointers to a linked |
   |               list of pointers to objects                |
   |                                                          |
   | volume  -->   contains  bounding  volume  for  each  of the four |
   |               primitive types in local unit coordinates  -  |
   |                                                          |
   |   cylinder    rectangular box,  back face  centered  at  origin |
   |               XYZ dimensions = 2 x 2 x 1                 |
   |                                                          |
   |   sphere      rectangular  box,  centered  at  origin  XYZ |
   |               dimensions = 2 x 2 x 2                     |
   |                                                          |
   |   cone        pyramid, apex at origin                    |
   |               length = 1,  base = 1 x 1                  |
   |                                                          |
   |   cube        unit cube along X, Y, and Z axes  (bounding volume |
   |               for cube is itself a cube)                 |
   |                                                          |
   ============================================================ */
```

```
char *
        nameof[4]     = {"cube","sphere","cylinder","cone" } ,


int
        dither4[4][4] = {
                         0,  8,  2, 10,
                        12,  4, 14,  6,
                         3, 11,  1,  9,
                        15,  7, 13,  5
                        },

        dither8[8][8] = {
                         0, 32,  8, 40,  2, 34, 10, 42,
                        48, 16, 56, 24, 50, 18, 58, 26,
                        12, 44,  4, 36, 14, 46,  6, 38,
                        60, 28, 52, 20, 62, 30, 54, 22,
                         3, 35, 11, 43,  1, 33,  9, 41,
                        51, 19, 59, 27, 49, 17, 57, 25,
                        15, 47,  7, 39, 13, 45,  5, 37,
                        63, 31, 55, 23, 61, 29, 53, 21 },

double
        xfacvw, yfacvw,         /* viewport -> window */
        xfacwv, yfacwv          /* window -> viewport */

NODE *
        grid[GRIDROW][GRIDCOL],


POINT
        volume[4][8] =  {
                          0, 0, 0,  1, 0, 0,      /*  cube  */
                          1, 1, 0,  0, 1, 0,
                          0, 0, 1,  1, 0, 1,
                          1, 1, 1,  0, 1, 1,

                         -1,-1,-1,  1,-1,-1,      /*  sphere  */
                          1, 1,-1, -1, 1,-1,
                         -1,-1, 1,  1,-1, 1,
                          1, 1, 1, -1, 1, 1,

                         -1,-1, 0,  1,-1, 0,      /* cylinder */
                          1, 1, 0, -1, 1, 0,
                         -1,-1, 1,  1,-1, 1,
                          1, 1, 1, -1, 1, 1,

                          0, 0, 0,  0, 0, 0,      /*  cone  */
                          0, 0, 0,  0, 0, 0,
                         -1,-1, 1,  1,-1, 1,
                          1, 1, 1, -1, 1, 1
                        },

/* UTILITY FUNCTIONS
   ================================================================
  |                                                                |
  | preprocess          process list of objects prior to raytracing, |
  |                     calculating inverse matrices, extents etc  |
  |                                                                |
  | transformvector     applies a transform matrix to vector       |
  |                                                                |
  | transformpoint      applies a transform matrix to point        |
  |                                                                |
  | generateray         maps given pixel (X Y)  to  window  and    |
  |                     generates ray equation                     |
```

```
  |                                                                    |
  | nextobject          used  by raycast to determine the next object  |
  |                     to be tested for intersection                  |
  |                                                                    |
  | normalize           converts a vector to unit form                 |
  |                                                                    |
  | linecompress        compresses image file size                     |
  |                                                                    |
   ====================================================================
*/

void preprocess(scene)
OBJECT * *scene,

{OBJECT * o,
POINT vol[8],

 if (POINTSOURCE == OFF) normalize(light),

 xfacwv = (double)(vx2-vx1)/(wx2-wx1),
 if (xfacwv < 0 0) xfacwv = -xfacwv,
 yfacwv = (double)(vy2-vy1)/(wy2-wy1),
 if (yfacwv < 0 0) yfacwv = -yfacwv,

 xfacvw = 1 0/xfacwv,
 yfacvw = 1 0/yfacwv,


 for (o=*scene, o!=NULL, o=o->next)
  {inversematrix(o->transform,o->inverse),
   if ((EXTENTS == ON) || (SORTLIST == ON) ||
       (GRID == ON)    || (PIXELBUFFER == ON))
      {transformvolume(o->transform,volume[(int)o->primitive],vol),
       if (SORTLIST == ON) calczdepth(vol,o),
       if ((PIXELBUFFER == ON) || (EXTENTS == ON) || (GRID == ON))
         projectvolume(vol),
       if ((PIXELBUFFER == ON) && (PGA == ON)  && (SCREEN == ON))
         rendervolume(vol,fillcolor),
       if (EXTENTS == ON)  calcextent(vol,o),
      }
   }

 if (SORTLIST == ON) *scene = sortlst(*scene,compare),
 if (GRID == ON)     makegrid(*scene),
}


void transformvector(m,vec,invvec)
MATRIX m,
VECTOR vec,invvec,
/*
   ====================================================================
  | Applies transformation matrix specified in m, to specified         |
  | vector, vec, returning result in invec, the transformed vector     |
   ====================================================================
*/
{
  invvec[0] = vec[0]*m[0][0] + vec[1]*m[1][0] + vec[2]*m[2][0] ,
  invvec[1] = vec[0]*m[0][1] + vec[1]*m[1][1] + vec[2]*m[2][1] ,
  invvec[2] = vec[0]*m[0][2] + vec[1]*m[1][2] + vec[2]*m[2][2] ,
}



void transformpoint(m,pt,invpt)
MATRIX m,
POINT pt,invpt,
```

```
/*
   ==========================================================
  | Applies transformation matrix specified in m, to specified |
  | point, pt, returning result in invpt, the transformed point |
   ==========================================================
*/
{
  invpt[0] = pt[0]*m[0][0] + pt[1]*m[1][0] + pt[2]*m[2][0] + m[3][0],
  invpt[1] = pt[0]*m[0][1] + pt[1]*m[1][1] + pt[2]*m[2][1] + m[3][1],
  invpt[2] = pt[0]*m[0][2] + pt[1]*m[1][2] + pt[2]*m[2][2] + m[3][2],
}



void generateray(x,y,p,d)
int x,y,                              /* pixel screen coordinates     */
POINT p,                              /* calculated ray origin        */
VECTOR d,                             /* calculated ray direction     */
/*
   ==========================================================
  | Generates the equation of a ray through the specified pixel, |
  | (x,y), as a point, p, and direction vector, d                |
   ==========================================================
*/
{
  p[0] = (x - vx1) * xfacvw + wx1,     /* map pixel to window in   */
  p[1] = (y - vy1) * yfacvw + wy1,     /* XY plane                 */
  p[2] = viewplanedist,

  if (PERSPECTIVE == OFF)              /*   zero  => parallel view */
   {
     d[0] = projection[0],             /* parallel view ==> read   */
     d[1] = projection[1],             /* specified direction      */
     d[2] = projection[2],
     }
  else                              /* defined => perspective view     */
   {
    d[0] = p[0] - viewpoint[0],    /* perspective view ==> calculate */
    d[1] = p[1] - viewpoint[1],    /* direction from view point and  */
    d[2] = p[2] - viewpoint[2],    /* mapped pixel point on window   */
    p[0] = viewpoint[0],
    p[1] = viewpoint[1],
    p[2] = viewpoint[2],
    }
}



void normalize(v)
VECTOR v,                             /* vector to be normalized */
/*
   ==========================================================
  | Takes vector of arbitrary length, v, and overwrites it with the |
  | calculated unit vector in the same direction                 |
   ==========================================================
*/

{double modl,

 modl = sqrt(v[0]*v[0] + v[1]*v[1] + v[2]*v[2]),   /* vector length */
 v[0] /= modl,  v[1] /= modl,  v[2] /= modl,       /* unit vector   */
}
```

```
OBJECT * nextobject(o,n)
OBJECT *o,
NODE   **n,
/*
   ============================================================
  | This function is used by raycast to determine the next object to |
  | test for intersection  If the grid optimization is in use,  the |
  | next  object  is  found from the linked list of pointers for the |
  | grid,  otherwise the next object is simply the next one  in  the |
  | object list                                                      |
   ============================================================
*/
{
  if (GRID == OFF) return o->next,
  if ((*n) == NULL)  return NULL,

  *n = (*n)->next,
  return  (*n)->ptr,
}



int linecompress(bufin,bufout,y,x1,x2)
unsigned char bufin[],                  /* line to be encoded        */
             bufout[],                  /* encoded line              */
int y,x1,x2,                            /* screen line number and    */
                                        /* start & end pixel numbers */
/*
   ============================================================
  | This function compresses a line of pixels  using  a  run  length |
  | encoding  system  compatible with that of the PGA   Each line of |
  | the image is preceeded by the hexadecimal code D9,  which is the |
  | PGA code for 'write encoded line of pixels'  Then follows three |
  | integer  numbers,  each  stored  as two bytes (least significant |
  | byte first),  representing the line number and the pixel numbers |
  | where this line begins and ends ie   -                           |
  |                                                                  |
  |         D9  line #   start x   end x       data                  |
  |                                                                  |
  | Using this format,  the image file can be sent directly  to  the |
  | PGA  without  any  processing  whenever  the  image  needs to be |
  | displayed from the file                                          |
  |                                                                  |
  | The data consists of packets, of which there are two kinds  -    |
  |                                                                  |
  |    [1]  COUNT  PEL                        { COUNT   0  127 }     |
  |    [2]  COUNT  PEL0 PEL1 PEL2             { COUNT 128  255 }     |
  |                                                                  |
  | If  the  count is in the range 0  127 then the byte that follows |
  | is the color of the next COUNT+1 pixels                          |
  |                                                                  |
  | If the count is in the range 128  255 (ie   MSB = 1)  then  the |
  | next  COUNT-127  bytes  that  follow  are the colors of the next |
  | COUNT-127 pixels                                                 |
  |                                                                  |
   ============================================================
*/
{unsigned char *p,*t,count,
int ptr=0,len,

  len = x2 - x1 + 1,
  p   = bufin,
  bufout[ptr++] = 0xd9,                 /* PGA code          */
  bufout[ptr++] = y & 0xff,             /* line # LSB        */
  bufout[ptr++] = (y & 0xff00) >> 8,    /* line # MSB        */
  bufout[ptr++] = x1 & 0xff,            /* start pixel LSB   */
  bufout[ptr++] = (x1 & 0xff00) >> 8,   /* start pixel MSB   */
```

```
  bufout[ptr++] = x2 & 0xff,                    /* end pixel LSB      */
  bufout[ptr++] = (x2 & 0xff00) >> 8,           /* end pixel MSB      */

  while (len > 1)
   {count = 0,
    if (*p == *(p+1))                           /* run of same color  */
     {count += 1,
      len -= 2,
      p += 2,
      while ((len > 0) && (count < 127) && (*p == *(p-1)))
       {count++,
        len--,
        p++,
         }
      bufout[ptr++] = count,
      bufout[ptr++] = *(p-1),
      }
    else
     {t = p,                                    /* run of different color */
      count += 1,
      len -= 2,
      p += 2,
      while ((len > 0) && (count < 127) &&
            (*p != *(p-1)) && (*p != *(p+1)))
        {len--,
         count++,
         p++,
         }
       bufout[ptr++] = count + 128,
       for (; t<p; t++)
        bufout[ptr++] = *t,
       }
    }
  if (len > 0)
    {bufout[ptr++] = 0,
     bufout[ptr++] = *p,
     }
  return ptr,
}

/* RAYTRACER
   ==========================================================
  | The following are the principal functions of the ray tracer - |
  |                                                               |
  |  inshadow      Checks if a given point lies in shadow         |
  |                                                               |
  |  testray       Tests a specified object for intersection with a |
  |                given ray                                      |
  |                                                               |
  |  raycast       Casts a ray through a specified pixel into the |
  |                scene of objects, tests for intersection and   |
  |                returns the appropiate color for the pixel     |
  |                                                               |
  |* pgatrace      Passes pixels in top to bottom, left to right  |
  |                fashion to raycast and collects/coordinates the |
  |                returned pixel intensity values to build up a  |
  |                screen and/or file image for the PGA adapter   |
  |                monitor                                        |
  |                                                               |
  |  rgbtrace      Same as pgatrace except that pixels are        |
  |                calculated as three seperate intensities (red, |
  |                green & blue)  Produces only a file output     |
  |                                                               |
  |  raytrace      initializes the raytracer and calls either     |
  |                pgatrace or rgbtrace to generate the image     |
  |                                                               |
  |           * function code in PGADPEND C module                |
   ========================================================== */
```

```
int inshadow(scene,obj,pt,dirn,tl)
OBJECT *obj,*scene,                    /* list of objects            */
POINT pt,                              /*  pt & dirn --> ray equation */
VECTOR dirn,
double tl,                             /* upper limit for t          */
/*
   ============================================================
  | Checks  if  a  point  on an object lies in shadow by testing the |
  | given shadow ray,  specified by pt & dirn and originating  at  a |
  | point on the object pointed to by obj, for intersection with the |
  | list  of  objects  pointed to by scene   An upper limit for t is |
  | specified in tl,  since objects beyond the light source need not |
  | be  tested                                                       |
   ============================================================
*/

{OBJECT *o,
POINT invpt,
VECTOR invdirn,
int shadow=0,          /* break as soon as any intersection found   */

  for (o=scene, ((o'=NULL) && (shadow'=1)), o=o->next)
   if (o '= obj)           /* don't want to test intersected object */
   {
    transformpoint(o->inverse,pt,invpt),          /* transform ray */
    transformvector(o->inverse,dirn,invdirn),

    switch (o->primitive) {               /* test for intersection */

            case cube       shadow = stracecube(invpt,invdirn,tl),
                            break,

            case sphere     shadow = stracesphere(invpt,invdirn,tl),
                            break,

            case cylinder   shadow = stracecylinder(invpt,invdirn,tl),
                            break,

            case cone       shadow = stracecone(invpt,invdirn,tl),
                            break,
                          }

       }

    return shadow,      /* shadow=1 if intersection with any object */
}


void testray(o,pt,dirn,nrml,dst)
OBJECT *o,                          /* pointer to object to test     */
POINT pt,                           /* pt & dirn --> ray equation    */
VECTOR dirn,nrml,                   /* nrml  returned surface normal */
double *dst,                        /* minimum t value               */
/*
   ============================================================
  | Tests the given ray,  specified in pt & dirn,  for  intersection |
  | with  the  object pointer to by o   If an intersection is found, |
  | which is closer than the t value specified in dst,  the  surface |
  | normal at the point of intersection is returned in nrml, and dst |
  | is updated                                                       |
   ============================================================
*/
{POINT invpt, invdirn,

   transformpoint(o->inverse,pt,invpt),          /* transform ray */
   transformvector(o->inverse,dirn,invdirn)
```

```
  switch (o->primitive) {

          case cube       tracecube(invpt,invdirn,nrml,dst),
                          break,

          case sphere     tracesphere(invpt,invdirn,nrml,dst),
                          break,

          case cylinder   tracecylinder(invpt,invdirn,nrml,dst),
                          break,

          case cone       tracecone(invpt,invdirn,nrml,dst),
                          break,
                      }
}


void raycast(scene,x,y,c)
OBJECT *scene,                    /* pointer to list of objects    */
int x,y,                          /* current pixel                 */
char * c,                         /* returned pixel color          */

{double  dst=INFINITY,t1,
         nearest=INFINITY,
         Zn,
OBJECT *closest=NULL,*o,
int row,col,shad=0,
NODE *n,

POINT  pt,dirn,
       nrml,invnrml,
       lght,interst,

generateray(x,y,pt,dirn),         /* generate ray equation for pixel */

if (SORTLIST == ON)
  Zn  = pt[2] + nearest*dirn[2], /* Z coord of nearest intersection */

if (GRID == ON)
 {                                /* calculate rectangle in   */
  row = (vy2-y)/((vy2-vy1)/GRIDROW+1), /* which pixel lies      */
  col = (x-vx1)/((vx2-vx1)/GRIDCOL+1),

  if ((n=grid[row][col]) == NULL)      /* any objects in this the  */
   {*c = background,                   /* rectangle ?              */
    o = NULL,
    }
   else o = n->ptr,
  }
 else o = scene,

 while (o != NULL)                        /* test list of objects  */
    {
     if ((EXTENTS == ON) &&
          ((x>o->xmax) || (x<o->xmin) || (y>o->ymax) || (y<o->ymin)))
        {o = nextobject(o,&n),
         continue,               /* pixel outside extent of current */
         }                       /* object, so skip object          */
     if ((SORTLIST == ON) && (o->znear < Zn))
       break,
      testray(o,pt,dirn,nrml,&dst),
      if (dst < nearest)
        {nearest = dst                     /* closer intersection   */
         closest = o,
         if (SORTLIST == ON)
           Zn  = pt[2] + nearest*dirn[2],
         }
```

```
      o = nextobject(o,&n),
     }

  if (closest != NULL)
   {
    transformvector(closest->transform,nrml,invnrml),
    if ((POINTSOURCE == ON) || (SHADOWS == ON))
       {interst[0] = pt[0] + nearest*dirn[0],
        interst[1] = pt[1] + nearest*dirn[1],
        interst[2] = pt[2] + nearest*dirn[2],
        }
    normalize(invnrml),
    normalize(dirn),

    if (POINTSOURCE == ON)
       {lght[0] = light[0] - interst[0],
        lght[1] = light[1] - interst[1],
        lght[2] = light[2] - interst[2],
        normalize(lght),
        if (SHADOWS == ON)
          { if (lght[0] != 0) tl = (light[0] - interst[0])/lght[0],
              else
               if (lght[1] != 0) tl = (light[1] - interst[1])/lght[1],
                else
                 if (lght[2] != 0) tl = (light[2] - interst[2])/lght[2],
            }
        }
      else
       { lght[0] = light[0],
         lght[1] = light[1],
         lght[2] = light[2],
         if (SHADOWS == ON) tl = INFINITY,
         }

     if (SHADOWS == ON)
         shad = inshadow(scene,closest,interst,lght,tl),

     if (PGA == ON)  pgashade(c,closest,dirn,lght,invnrml,x,y,shad),
      else           rgbshade(c,closest,dirn,lght,invnrml,x,y,shad),
     }
  else *c = backgrounnd,
}


void rgbtrace(scene)
OBJECT *scene,

{char *bf, *ptr, id,
int x,y,

bf = malloc((3*vx2-vx1+1)),

if (FILEOUT == ON)
    {id = 0x00,              /* write id byte to indicate RGB format */
     fwrite((char *)&id,1,1,outfile),
     fwrite((char *)&vx1,sizeof(int),1,outfile),   /* write viewport */
     fwrite((char *)&vx2,sizeof(int),1,outfile),
     fwrite((char *)&vy1,sizeof(int),1,outfile),
     fwrite((char *)&vy2,sizeof(int),1,outfile),
     }

 for (y=vy2, y>=vy1, y--)
  {ptr = bf
   for (x=vx1, x<=vx2, x++,ptr+=3)
     raycast(scene,x,y,ptr),
   if (FILEOUT == ON)
     fwrite(bf,3,vx2-vx1+1,outfile),
   }
```

```
  free(bf),
  fclose(outfile),
}


void raytrace(scene)
OBJECT * *scene,

{

 preprocess(scene),
 if (PGA) pgatrace(*scene),
  else rgbtrace(*scene),
}
```

```
/* RAY INTERSECTION MODULE
   ========================================================
  | This module contains the ray-primitive intersection test |
  | functions  There are two sets of functions for each of the four |
  | primitive types, the first set for testing primary rays and the |
  | second for testing shadow rays                           |
  |                                                          |
  |      Primary Ray Functions          Secondary Ray Functions      |
  |                                                          |
  |         tracecylinder                    stracecylinder          |
  |         tracesphere                      stracesphere            |
  |         tracecube                        stracecube              |
  |         tracecone                        stracecone              |
  |                                                          |
   ========================================================
*/


#include   <math h>
#include   "typedef h"   /* structure and other typedef definitions*/
#include   "function h"  /* function prototype declarations       */


#define    EPSILON     0 00001


/* RAY PRIMITIVE INTERSECTION FUNCTIONS --- PRIMARY RAYS
   ========================================================
  | The following four functions are used to test a ray for |
  | intersection with one of the four primitive types implemented  |
  | Each one takes a ray as input, along with the lowest t value |
  | found in processing the list of primitives to date, and modifies |
  | the value if the ray intersects the primitive at a closer point  |
  | If such is the case, the normal to the surface at the point of |
  | intersection is returned by modifying the vector nrm, and the |
  | closest point of intersection, nearest, is updated       |
   ========================================================
*/


void tracecube(pt,dirn,nrm,nearest)
POINT pt,
VECTOR dirn, nrm,
double *nearest,
/*
   ========================================================
  | Tests given ray, specified by a point, pt, and a direction |
  | vector, dirn, for intersection with a unit cube defined by 6 |
  | planes -                                                 |
  |              X = 0   X = 1   Y = 0   Y = 1   Z = 0   Z = 1      |
   ========================================================
*/
{double  X,Y,Z,t,

 if (dirn[0] != 0)                          /* ray on x=0 plane ?    */
   { t = -pt[0]/dirn[0],                    /* if ray intersects X=0 */
     if ((t < *nearest) && (t > 0))         /* plane at positive t   */
      {Y = pt[1] + t*dirn[1],               /* value lower than the  */
       Z = pt[2] + t*dirn[2],               /* current closest one,  */
       if ((Y <= 1 0) && (Y >= 0 0)  &&     /* then check to see if  */
           (Z <= 1 0) && (Z >= 0 0))        /* it intersects face 0  */
         {*nearest = t                      /* of cube               */
          nrm[0] = -1 0,
          nrm[1] =  0 0,
          nrm[2] =  0 0,
          }
       }
```

```
    t = (1.0-pt[0])/dirn[0],              /* test X=1 plane */
    if ((t < *nearest)  && (t > 0))
     {Y = pt[1] + t*dirn[1],
      Z = pt[2] + t*dirn[2],
      if ((Y <= 1.0) && (Y >= 0.0)  &&
          (Z <= 1.0) && (Z >= 0.0))
        { *nearest = t,
          nrm[0] = 1.0,
          nrm[1] = 0.0,
          nrm[2] = 0.0,
         }
       }
   }
 if (dirn[1] != 0.0)                      /* test Y=0 plane */
   { t = -pt[1]/dirn[1],
    if ((t < *nearest)  && ( t > 0))
      {X = pt[0] + t*dirn[0],
       Z = pt[2] + t*dirn[2],
       if ((X <= 1.0) && (X >= 0.0)  &&
           (Z <= 1.0) && (Z >= 0.0))
         { *nearest = t,
           nrm[0] =  0.0,
           nrm[1] = -1.0,
           nrm[2] =  0.0,
          }
        }
     t = (1.0-pt[1])/dirn[1],             /* test Y=1 plane */
     if ((t < *nearest)  && (t > 0))
      {X = pt[0] + t*dirn[0],
       Z = pt[2] + t*dirn[2],
       if ((X <= 1.0) && (X >= 0.0)  &&
           (Z <= 1.0) && (Z >= 0.0))
         { *nearest = t,
           nrm[0] = 0.0,
           nrm[1] = 1.0,
           nrm[2] = 0.0,
          }
        }
     }

 if (dirn[2] != 0.0)                      /* test  Z=0 plane */
   { t = -pt[2]/dirn[2],
     if ((t < *nearest)   && (t > 0))
      {Y = pt[1] + t*dirn[1],
       X = pt[0] + t*dirn[0],
       if ((Y <= 1.0) && (Y >= 0.0)  &&
           (X <= 1.0) && (X >= 0.0))
         { *nearest = t,
           nrm[0] =  0.0,
           nrm[1] =  0.0,
           nrm[2] = -1.0,
          }
        }
     t = (1.0-pt[2])/dirn[2],             /* test Z=1 plane */
     if ((t < *nearest) && (t > 0))
      {Y = pt[1] + t*dirn[1],
       X = pt[0] + t*dirn[0],
       if ((Y <= 1.0) && (Y >= 0.0)  &&
           (X <= 1.0) && (X >= 0.0))
         { *nearest = t,
           nrm[0] = 0.0,
           nrm[1] = 0.0
           nrm[2] = 1.0,
          }
        }
     }
}
```

```
void tracesphere(pt,dirn,nrm,nearest)
POINT pt,
VECTOR dirn, nrm,
double *nearest,
/*
   ================================================================
  | Tests ray specified by pt and dirn for intersection with a  cube |
  | of  unit radius,  centered about origin,  by solving a quadratic |
  | equation for t,  obtained  by  substituting  ray  equation  into |
  | sphere equation  Equation is solved using the formula  -         |
  |                                                                  |
  |              t = (-B +/- sqrt(B*B - 4*A*C))/(2*A)                |
   ================================================================
*/
{double A,B,B_2,C,AC,BAC,t,

 A = dirn[0]*dirn[0] + dirn[1]*dirn[1] + dirn[2]*dirn[2],
 B = dirn[0]*pt[0] + dirn[1]*pt[1] + dirn[2]*pt[2],
 C = pt[0]*pt[0] + pt[1]*pt[1] + pt[2]*pt[2] - 1 0,
 B_2 = B*B,
 AC = A*C,
 BAC = B_2 - AC,
 if (BAC > 0 0)                          /* BAC < 0  ==> complex roots  */
  {                                      /* BAC = 0  ==> ray is tangent */
    t = (-B - sqrt(BAC))/A,
    if ((t < *nearest) && (t > 0))
     {*nearest = t,
      nrm[0] = pt[0] + t*dirn[0],
      nrm[1] = pt[1] + t*dirn[1],
      nrm[2] = pt[2] + t*dirn[2],
      }
   }
}


void tracecylinder(pt,dirn,nrm,nearest)
POINT pt,
VECTOR dirn,nrm,
double *nearest,
/*
   ================================================================
  | A  ray,  specified by pt and dirn,  is tested for intersection a |
  | cylinder of unit length and  unit  radius,  centered  along  the |
  | positive  z axis,  with its back face centered about the origin  |
  | Three seperate tests are performed, one for each of  -           |
  |                                                                  |
  | the back  face     unit circle on Z=0 plane, centre at (0,0,0)   |
  | the front face     unit circle on Z=1 plane, centre at (0,0,1)   |
  | the main  body     tube of unit radius centered along Z axis     |
   ================================================================
*/
{double A,B,B_2,C,AC,BAC,t,X,Y,Z,

 if (dirn[2] != 0 0)                     /* test back face */
  { t = -pt[2]/dirn[2],
    if ((t < *nearest) && (t > 0))       /* closer intersection with    */
     {                                   /* Z=0 plane ?                 */
       X = pt[0] + t*dirn[0],            /* if yes, calculate point     */
       Y = pt[1] + t*dirn[1],            /* of intersection             */
       if ((X*X + Y*Y) <= 1 0)           /* Does point lie on back face */
         { *nearest = t,
           nrm[0] =  0 0,
           nrm[1] =  0 0,
           nrm[2] = -1 0,
           }
       }
```

```
    t = (1 0-pt[2])/dirn[2],          /* test front face in same way */
    if ((t < *nearest) && (t > 0))
     {
       X = pt[0] + t*dirn[0],
       Y = pt[1] + t*dirn[1],
       if ((X*X + Y*Y) <= 1 0)
         { *nearest = t,
           nrm[0] = 0 0,
           nrm[1] = 0 0,
           nrm[2] = 1 0,
           }
         }
     }
  A = dirn[0]*dirn[0] + dirn[1]*dirn[1],     /* test main body by   */
  B = dirn[0]*pt[0] + dirn[1]*pt[1],         /* solving a quadratic */
  C = pt[0]*pt[0] + pt[1]*pt[1] - 1 0,       /* equation for t      */
  B_2 = B*B,
  AC = A*C,
  BAC = B_2 - AC,

 if ((BAC > 0 0) && (A != 0 0))             /* BAC < 0 => complex root */
 {                                          /* BAC = 0 => ray tangent  */
    t  = (-B - sqrt(BAC))/A,
    if ((t < *nearest) && (t > 0))
      {Z = pt[2] + t*dirn[2],
       if ((Z <= 1 0) && (Z >= 0 0))
        { *nearest = t,
          nrm[0] = pt[0] + t*dirn[0],
          nrm[1] = pt[1] + t*dirn[1],
          nrm[2] = 0,
          }
        }
    }
}

void tracecone(pt,dirn,nrm,nearest)
POINT pt,
VECTOR dirn, nrm,
double *nearest,
/*
   ================================================================
  | Specified ray,  defined by pt and dirn,  is tested  for        |
  | intersection with a cone of unit length, apex at origin, and   |
  | base of unit radius, centre (0,0,1)  Two seperate tests are    |
  | performed, one for the base and one for the main body          |
   ================================================================
*/
{double A,B,B_2,C,AC,BAC,t,X,Y,Z,

 A = dirn[0]*dirn[0] + dirn[1]*dirn[1] - dirn[2]*dirn[2],
 B = dirn[0]*pt[0] + dirn[1]*pt[1] - pt[2]*dirn[2]
 C = pt[0]*pt[0] + pt[1]*pt[1] - pt[2]*pt[2],
 B_2 = B*B,
 AC = A*C,                                  /* test with main body by  */
 BAC = B_2 - AC,                            /* solving quadratic fot t */
 if ((BAC > 0 0) && (A != 0 0))
  {
    t = (-B - sqrt(BAC))/A,
    if ((t < *nearest) && (t > 0))
     {Z = pt[2] + t*dirn[2],
      if ((Z <= 1 0) && (Z >= 0 0))
        {*nearest = t,
          nrm[0] =  pt[0] + t*dirn[0],
          nrm[1] =  pt[1] + t*dirn[1],
          nrm[2] =  sqrt(1 0 - Z*Z),
         }
       }
  }
```

```
 if (dirn[2] != 0 0)                    /* test base, same test as */
  { t = (1 0-pt[2])/dirn[2],            /* front face of cylinder  */
    if ((t < *nearest) && (t > 0))
     {X = pt[0] + t*dirn[0],
      Y = pt[1] + t*dirn[1],
      if ((X*X + Y*Y) <= 1 0)
       { *nearest = t,
           nrm[0] = 0 0,
           nrm[1] = 0 0,
           nrm[2] = 1 0,
         }
      }
    }
}

/* RAY-PRIMITIVE INTERSECTION FUNCTIONS --  SHADOW RAYS
   ============================================================
  | These four functions below are used to test shadow rays ie  |
  | rays traced from a point on an object to the light source, for |
  | intersection with one of the four primitive solid types  The |
  | functions are very similar to the primary ray intersection |
  | functions above but differ in the respect that they do not need |
  | to determine the closest surface of a primitive struck by the |
  | ray, only if the ray strikes any surface between the ray origin |
  | and the light source ie  0 < t < tl where tl is the upper limit |
  | determined from the distance of the ray origin to the light |
  | source  The return value is 0 if there was no intersection, 1 if |
  | there was                                                    |
   ============================================================
*/

int stracecube(pt,dirn,tl)
POINT pt,
VECTOR dirn,
double tl,
{double X,Y,Z,t,

 if (dirn[0] != 0)                      /* test X+0 plane */
   { t = -pt[0]/dirn[0],
     if ((t > EPSILON) && (t< tl))      /* allow for roundoff error */
       {Y = pt[1] + t*dirn[1],
        Z = pt[2] + t*dirn[2],
        if ((Y <= 1 0) && (Y >= 0 0)  &&
            (Z <= 1 0) && (Z >= 0 0))
          return 1,
        }

     t = (1 0-pt[0])/dirn[0],           /* test X=1 plane */
     if ((t > EPSILON) && (t < tl))
       {Y = pt[1] + t*dirn[1],
        Z = pt[2] + t*dirn[2],
        if ((Y <= 1 0) && (Y >= 0 0)  &&
            (Z <= 1 0) && (Z >= 0 0))
          return 1,
         }
        }

 if (dirn[1] != 0)                      /* test Y=0 plane */
   { t = -pt[1]/dirn[1],
     if ((t > EPSILON) && (t < tl))
       {X = pt[0] + t*dirn[0]
        Z = pt[2] + t*dirn[2],
        if ((X <= 1 0) && (X >= 0 0)  &&
            (Z <= 1 0) && (Z >= 0 0))
          return 1,
        }
```

```
    t = (1.0-pt[1])/dirn[1],          /* test Y=1 plane */
    if ((t > EPSILON) && (t < tl))
      {X = pt[0] + t*dirn[0],
       Z = pt[2] + t*dirn[2],
       if ((X <= 1.0) && (X >= 0.0)  &&
           (Z <= 1.0) && (Z >= 0.0))
         return 1,
       }
    }

 if (dirn[2] != 0.0)                  /* test Z=0 plane */
   { t = -pt[2]/dirn[2],
     if ((t > EPSILON) && (t < tl))
       {Y = pt[1] + t*dirn[1],
        X = pt[0] + t*dirn[0],
        if ((Y <= 1.0) && (Y >= 0.0)  &&
            (X <= 1.0) && (X >= 0.0))
          return 1,
        }
     t = (1.0-pt[2])/dirn[2],          /* test Z=1 plane */
     if ((t > EPSILON) && (t < tl))
       {Y = pt[1] + t*dirn[1],
        X = pt[0] + t*dirn[0],
        if ((Y <= 1.0) && (Y >= 0.0)  &&
            (X <= 1.0) && (X >= 0.0))
          return 1,
        }
     }
  return 0,
}

int stracesphere(pt,dirn,tl)
POINT pt,
VECTOR dirn,
double tl,
{double A,B,B_2,C,AC,BAC,t,

 A = dirn[0]*dirn[0] + dirn[1]*dirn[1] + dirn[2]*dirn[2],
 B = dirn[0]*pt[0] + dirn[1]*pt[1] + dirn[2]*pt[2],
 C = pt[0]*pt[0] + pt[1]*pt[1] + pt[2]*pt[2] - 1.0,
 B_2 = B*B,
 AC = A*C,
 BAC = B_2 - AC,
 if (BAC > 0.0)
  {
     t = (-B - sqrt(BAC))/A,
      if ((t > EPSILON) && (t < tl)) return 1,
      }
  return 0,
}


int stracecylinder(pt,dirn,tl)
POINT pt,
VECTOR dirn,
double tl,

{double A,B,B_2,C,AC,BAC,t,X,Y,Z,

 if (dirn[2] != 0.0)                          /* test back face */
  { t = -pt[2]/dirn[2],
    if ((t > EPSILON) && (t < tl))
      {X = pt[0] + t*dirn[0],
       Y = pt[1] + t*dirn[1],
       if ((X*X + Y*Y) <= 1.0)  return 1,
       }
```

```
    t = (1.0-pt[2])/dirn[2],                    /* test front face */
    if ((t > EPSILON) && (t < t1))
      {X = pt[0] + t*dirn[0],
       Y = pt[1] + t*dirn[1],
       if ((X*X + Y*Y) <= 1.0)  return 1,
       }
     }

  A = dirn[0]*dirn[0] + dirn[1]*dirn[1],        /* test main body */
  B = dirn[0]*pt[0] + dirn[1]*pt[1],
  C = pt[0]*pt[0] + pt[1]*pt[1] - 1.0,
  B_2 = B*B,
  AC = A*C,
  BAC = B_2 - AC,

 if ((BAC > 0.0) && (A != 0.0))
  {
     t  = (-B - sqrt(BAC))/A,
     if ((t <= 0) || (t >= t1)) return 0,
     Z = pt[2] + t*dirn[2],
     if ((Z <= 1.0) && (Z >= 0.0))  return 1,
    }
  return 0,
}


int stracecone(pt,dirn,t1)
POINT pt,
VECTOR dirn,
double t1,

{double A,B,B_2,C,AC,BAC,t,X,Y,Z, -   --        --   - - - - - -  - -

 if (dirn[2] != 0.0)                            /* test base */
  { t = (1.0-pt[2])/dirn[2],
    if ((t > EPSILON) && (t < t1))
      {X = pt[0] + t*dirn[0],
       Y = pt[1] + t*dirn[1],
       if ((X*X + Y*Y) <= 1.0)  return 1,
       }
    }

 A = dirn[0]*dirn[0] + dirn[1]*dirn[1] - dirn[2]*dirn[2],
 B = dirn[0]*pt[0] + dirn[1]*pt[1] - pt[2]*dirn[2],
 C = pt[0]*pt[0] + pt[1]*pt[1] - pt[2]*pt[2],
 B_2 = B*B,
 AC = A*C,
 BAC = B_2 - AC,                                /* test main body */
 if ((BAC > 0.0) && (A != 0.0))
  {
     t = (-B - sqrt(BAC))/A,
     if ((t <= 0) || (t >= t1)) return 0,
     Z = pt[2] + t*dirn[2],
     if ((Z <= 1.0) && (Z >= 0.0))
      return 1,
    }
  return 0,
}
```

```
/* OPTIMIZING FUNCTIONS MODULE
   ============================================================
  | This module contains the functions used to implement the |
  | pixelbuffer, sortlist, extent and grid optimizations     |
  |                                                          |
  |       Utility Functions          Optimizing Functions    |
  |                                                          |
  |         transformvolume                calcextent        |
  |         projectvolume                  sortlst           |
  |         getnode                        makegrid          |
  |         compare                      * rendervolume      |
  |         calczdepth                                       |
  |                                                          |
  |     * contained in PGA dependent module  PGADPEND C      |
   ============================================================
*/

#include   <stdio h>
#include   <malloc h>
#include   "global h"     /* global parameters, flags & variables  */
#include   "function h"   /* function prototype declarations       */

/* UTILITY FUNCTIONS
   ============================================================
  |                                                          |
  | transformvolume     Apply transform matrix of an instance of a |
  |                     primitive to its bounding volume     |
  |                                                          |
  | projectvolume       Project transformed bounding volume onto |
  |                     viewing plane                        |
  |                                                          |
  | getnode             Return pointer to allocated space for a |
  |                     structure of type NODE               |
  |                                                          |
  | compare             Used by the sortlst function in sorting the |
  |                     list of objects  Compares two objects, A and |
  |                     B on the basis of the value of their |
  |                     respective znear fields              |
  |                                                          |
  | calczdepth          Used to calculate the nearest & furthest z |
  |                     coords of transformed bounding volumes |
  |                                                          |
   ============================================================
*/


void transformvolume(m,v,r)
MATRIX m,
POINT v[8],r[8],
/*
   ============================================================
  | Takes specified local bounding volume, v,  and applies specified |
  | transform matrix,  m,  returning transformed volume in r   (each |
  | volume is defined as array of eight 3D vertices)         |
   ============================================================
*/
{int i,j,k,

 for (i=0, i<8, i++)              /*   r_vertex = v_vertex * m    */
   for (j=0, j<3, j++)
    { r[i][j] = 0,
      for (k=0, k<3, k++)
       r[i][j] += v[i][k] * m[k][j]
     r[i][j] += m[3][j],
     }                                      /* implied r[i][4] of 1 */

}
```

```
void projectvolume(v)
POINT  v[8],
/*
   ==========================================================
  | Takes specified (transformed) bounding volume and overwrites it |
  | with its projectection onto the viewing plane                   |
   ==========================================================
*/
{int i,j,
 double sx,sy,sz,              /* X and Y scale factors  */

if (PERSPECTIVE == OFF)
 {
  if (projection[2] != 0)
    { sx = projection[0]/projection[2],
      sy = projection[1]/projection[2],
     }
   else
     { sx = INFINITY,              /* parallel projection along the  */
       sy = INFINITY,              /* direction D(xd,yd,zd) of a     */
                                   /* point P(x,y,z) to P'(xv,yv,zv) */
      }                            /* on the plane Z=zv is  -        */
                                   /*                                */
   for (i=0, i<8, i++)             /*      xv = x + xd*(zv-z)/zd     */
     {                             /*      yv = y + yd*(zv-z)/zd     */
       v[i][0] += (viewplanedist-v[i][2]) * sx,
       v[i][1] += (viewplanedist-v[i][2]) * sy,
       v[i][2]  = viewplanedist,
      }
   }
else
   { sz = viewplanedist - viewpoint[2],
     for (i=0, i<8, i++)
      {sx = sy = v[i][2] - viewpoint[2],
       if (sx != 0)
         {v[i][0] = viewpoint[0] + (v[i][0]-viewpoint[0])*sz/sx,
          v[i][1] = viewpoint[1] + (v[i][1]-viewpoint[1])*sz/sx,
          }
        else  v[i][0] = v[i][1] = INFINITY,
        v[i][2]  = viewplanedist,
       }
     }
}


NODE *getnode(i,j)
int i,j,
/*
   ==========================================================
  | Used  by  the  makegrid  function  to  allocate space for nodes  |
   ==========================================================
*/
{NODE *p,
  return (NODE *) malloc(sizeof(NODE)),
}


int compare(a,b)
OBJECT *a,*b,
/*
   ==========================================================
  | Compares two objects,  a and b,  on the  basis  of  their  znear |
  | fields  (the  nearest  Z  coordinate  of the objects transformed |
  | bounding volume)  Used by the sortlist  function to sort the list |
  | of objects into increasing distance from the viewer              |
   ==========================================================
*/
```

```
{
  if (a->znear == b->znear) return 0;
  if (a->znear <  b->znear) return 1;
  return -1;
}

void calczdepth(v,o)
POINT v[8];
OBJECT *o;
/*
   ================================================================
  | Takes specified transformed bounding volume,  v,  and calculates |
  | its nearest & furthest z coordinates,  placing them in the znear |
  | and zfar fields respectively,  of the object,  o.  Differs  from |
  | calcextent  function  in that volumes are not projected onto the |
  | view plane and  that  values  calculated  are  in  world  coords |
  | (double) rather than screen coords (int).                        |
   ================================================================
*/
{double z1,z2;                /* current near & far values   */
int i;

 z1 = z2 = v[0][2];            /* near = far = first verter  */

 for (i=1; i<8; i++)
    if (v[i][2] < z2) z2 = v[i][2];          /* check remaining 7  */
    else if (v[i][2] > z1) z1 = v[i][2];     /* vertices.          */

 if (z1 > 0) z1 += 1;
  o->znear = (int)(z1);
 if (z2 < 0) z2 -= 1;
 o->zfar = (int)(z2);
}


/* EXTENTS OPTIMIZATION
   ================================================================
  | The extents optimization works by calculating for each bject,  a |
  | minimal rectangular  area  on  the  screen  which  encloses  the |
  | objects  transformed  and  projected  bounding  volume.  The ray |
  | intersects bounding volume test  then  reduces  to  a  point  in |
  | rectangle  test,  namely  that the current pixel lies inside the |
  | objects screen rectangle/extent.  If it does not lie inside  the |
  | ray  spawned  by  the  pixel need not be tested for intersection |
  | with the object.                                                 |
  |                                                                  |
  | calcextent :        calculates  minimal  rectangle  on  screen,  |
  |                     enclosing projected bounding volume.         |
  |                                                                  |
   ================================================================
*/


void calcextent(v,o)
POINT v[8];
OBJECT *o;
/*
   ================================================================
  | Takes specified (projected) bounding volume,  v,  and calculates |
  | its minimal enclosing screen rectangle, defined by two XY screen |
  | coordinates  (xmin,ymin)  &  (xmax,ymax)  ,  the upper right and |
  | lower left corners,  and places it in the xmin,  xmax,  ymin and |
  | ymax fields of the specified object structure, o.                |
   ================================================================
*/
{double x1,x2,y1,y2,z1,z2;      /* current max & min X and Y values */
int i;
```

```
 x1 = x2 = v[0][0],              /* max = min = first vertex  */
 y1 = y2 = v[0][1],

 for (i=1, i<8, i++)
  { if (v[i][0] < x1) x1 = v[i][0],          /* check remaining 7  */
    else if (v[i][0] > x2) x2 = v[i][0],     /* vertices           */
    if (v[i][1] < y1) y1 = v[i][1],
    else if (v[i][1] > y2) y2 = v[i][1],
    }

  if (x1 < wx1) x1 = wx1,          /* clip rectangle to edge of   */
  if (x2 > wx2) x2 = wx2,          /* window if it exceeds same   */
  if (y1 < wy1) y1 = wy1,
  if (y2 > wy2) y2 = wy2,

  o->xmin = (x1 - wx1) * xfacwv + vx1,   /* map rectangle to screen  */
  o->xmax = (x2 - wx1) * xfacwv + vx1,
  o->ymin = (y1 - wy1) * yfacwv + vy1,
  o->ymax = (y2 - wy1) * yfacwv + vy1,
}


/* PIXEL BUFFER OPTIMIZATION
   ========================================================================
  | The  pixel  buffer  optimization works by taking the transformed |
  | and projected bounding volumes for all objects and drawing  them |
  | as  filled  polygons  on  the  screen,  all filled with the same |
  | color   By setting all screen pixels to a different color before |
  | performing  this  operation,  any pixel which is not of the fill |
  | color  can  be  instantly  identified  as  one  which  does  not |
  | intersect  any  object  at  all and can be set to the background |
  | color without even generating a ray equation for it   Since  the |
  | function  uses  PGA specific function calls to draw and fill the |
  | polygons, it can be found in the PGA module                      |
  |                                                                  |
  | rendervolume        render projected bounding volume  on  screen |
  |                     in fillcolor  (SEE PGA MODULE)               |
   ========================================================================
*/


/* SORTLIST OPTIMIZATION
   ========================================================================
  | The sortlist optimization works by taking  the  linked  list  of |
  | objects  (which  defines  the  scene) and sorting it in order of |
  | increasing bounding volume distance from  the  viewer  in  the   |
  | direction  of  projection  ie   the  closer  an objects bounding |
  | volume to the viewer (when transformed into  world  coordinates) |
  | the  nearer  the  front of the list that object is placed in the |
  | sorted list   Then, when testing a ray for intersection with the |
  | list of objects,  all objects whose closest bounding volume face |
  | lies further from the viewer than the intersection point  cannot |
  | possibly intersect the ray at a closer point, and can be omitted |
  | from the test   And,  since the list is in sorted order, as soon |
  | as one such object is encountered,  the test can be ended as all |
  | further  objects  in  the  list  must  lie even further from the |
  | viewer                                                           |
  |                                                                  |
  | sortlst      Takes a linked list and a pointer to a function for |
  |              comparing  two  nodes,  and  sorts  the  list  in   |
  |              ascending order,  returning a pointer  to  the  new |
  |              sorted list                                         |
   ========================================================================
*/

OBJECT *sortlst(p,compare)
OBJECT *p,
int (*compare)(),
```

```
{int base,
unsigned int block,
struct tape {
            OBJECT first,*last,
            unsigned int count,
             } tape[4],

tape[0] count = 0, tape[0] last = &tape[0] first,
tape[1] count = 0, tape[1] last = &tape[1] first,

for (base=0, p'=NULL, p=p->next, base^=1)
 {tape[base] last = tape[base] last->next = p,
  tape[base] count++,
  }

for (base=0, block=1, tape[base+1] count'=0, base ^=2, block <<=1)
 {int dest,
  struct tape *tape0, *tape1,
  tape0 = tape + base,
  tape1 = tape + base + 1,
  dest  = base^2,
  tape[dest] count = 0,  tape[dest] last = &tape[dest] first,
  tape[dest+1] count = 0, tape[dest+1] last = &tape[dest+1] first,

  for (, tape0->count'=0, dest^=1)
   {unsigned int n0,n1,
    struct tape *output_tape,
    output_tape = tape + dest,
    n0 = n1 = block,
    while (1)
     {OBJECT *chosen_item,
      struct tape *chosen_tape,
       if (n0==0 || tape0->count==0)
        {if (n1==0  || tape1->count==0) break,
         chosen_tape = tape1,
         n1--,
          }
       else if (n1==0  || tape1->count==0)
          {chosen_tape = tape0,
           n0--,
            }
       else if ((*compare)(tape0->first next,tape1->first next) > 0)
          {chosen_tape = tape1,
           n1--,
           }
        else
           {chosen_tape = tape0,
           n0--,
           }
       chosen_tape->count--,
       chosen_item = chosen_tape->first next,
       chosen_tape->first next = chosen_item->next,
       output_tape->last = output_tape->last->next = chosen_item,
       output_tape->count++,
        }
      }
     }
  tape[base] last->next = NULL,
  return tape[base] first next,
 }
```

```
/* GRID OPTIMIZATION
   ================================================================
  | The grid optimization works by dividing the screen up into a   |
  | fixed number of rectangles and associating with each a pointer |
  | to list of those objects whose screen extents overlap the      |
  | rectangle  Testing a ray for intersection with the scene then  |
  | involves determining in which rectangle the pixel spawning the |
  | ray lies and testing the ray only with the objects in the      |
  | rectangles associated list  The grid is implemented as a global|
  | 2 dimensional array of pointers  Each pointer points to a linked|
  | list whose nodes consist of two fields, a pointer to the next  |
  | node of the linked list and a pointer an object in the list of |
  | objects whose extent has overlapped the rectangle              |
  |                                                                |
  | makegrid    Takes a pointer to a list of objects and creates for|
  |             each pointer element of a global 2d array a linked |
  |             list of pointers to objects whose screen extents   |
  |             overlap the associated rectangle of the array      |
  |             pointer element                                    |
   ================================================================
*/

void makegrid(scene)
OBJECT *scene,

{int i,j,szx,szy,
OBJECT *p,
NODE *gp,

 szx = (vx2-vx1)/GRIDCOL + 1,
 szy = (vy2-vy1)/GRIDROW + 1,

 for (i=0, i<GRIDROW, i++)
  for (j=0, j<GRIDCOL, j++)
   grid[i][j] = NULL,

 for (i=0, i<GRIDROW, i++)
  for (j=0, j<GRIDCOL, j++)
   for (p=scene, p!=NULL, p=p->next)
    if ((p->xmin < (j+1)*szx+vx1) && (p->ymin <= vy2-i*szy) &&
        (p->xmax >= j*szx+vx1)    && (p->ymax > vy2-(i+1)*szy))
      {if (grid[i][j] == NULL)
         grid[i][j] = gp = getnode(i,j),
       else
         {gp->next = getnode(i,j),
          gp = gp->next,
          }
        if (gp != NULL)
         {gp->ptr = p,
          gp->next = NULL,
          }
         else
          {printf("\n\nOut of Space'     node %d %d",i,j),
           exit(0),
           }
       }
}
```

```
/* SHADING MODULE
   ==========================================================
   | This module contains the shading functions used by the |
   | raytracer  Although the pgashade function is geared to |
   | calculating a color in the PGA range 0  255, it does not call |
   | any pga library functions and so is omitted from the PGA |
   | dependent module PGADPEND C                            |
   |                                                        |
   | Functions -                                            |
   |              pgashade                                  |
   |              rgbshade                                  |
   |                                                        |
   ==========================================================
*/

#include    <math h>
#include    "typedef h"    /* structure & other typedef definitions*/
#include    "function h"   /* function prototype declarations      */


extern double    ambient,

extern int       DITHER, dither4[4][4], dither8[8][8],


/* SHADING FUNCTIONS
   ==========================================================
   | These functions use the following vectors to calculate the shade |
   | for  a given pixel,  using Phongs lighting equation (all vectors |
   | specified  in  relation  to  intersection  point  of  ray  with |
   | primitive object and are assumed to be unit vectors)  -  |
   |   -                                                    |
   |      ray   -->  vector in direction of viewer/ray      |
   |      lght  -->  vector in direction of light source    |
   |      nrml  -->  surface normal                         |
   |                                                        |
   | Other  information  is accessed via a pointer to the intersected |
   | object,  obj   Pixel coordinates are passed through x and  y  in |
   | case  a  dither matrix operation is specified through the global |
   | flag 'DITHER'                                          |
   ==========================================================
*/


void pgashade(c,obj,ray,lght,nrml,x,y,shad)
unsigned char *c,
OBJECT *obj,
VECTOR ray,lght,nrml,
int x,y,shad,
/*
   ==========================================================
   | This function calculates the shade of a pixel  as  an  intensity |
   | value in the range 0  1   This is then converted to an int value |
   | in the range 0  15   Since the raytracer loads the PGA 256 entry |
   | color table with 16 different shades for each of 16 base colors, |
   | this value is added to the offset of the base color in the table |
   | to give a value in the range 0  255                    |
   ==========================================================
*/
{VECTOR rflec,                  /* direction of reflected ray       */
double spec,pspec,              /* specular reflection values       */
       diff,                    /* diffuse reflection value         */
       intens,                  /* final calculated intensity       */
       rnd,
       dther,                   /* used in dither matrix operation  */
```

```
if (!shad)                          /* point not in shadow              */
 {
  diff = nrml[0]*lght[0] +                /* diffuse intensity =          */
         nrml[1]*lght[1] +                /* vector dot product of        */
         nrml[2]*lght[2];                 /* normal and light vectors     */

  if (diff < 0.0) diff = 0.0;             /* negative => angle > 90       */

  rflec[0] = lght[0]-ray[0];              /* calculate direction of       */
  rflec[1] = lght[1]-ray[1];              /* reflected ray for use in     */
  rflec[2] = lght[2]-ray[2];              /* specular calculation.        */

  normalize(rflec);                       /* must be unit vector          */

  if (diff == 0.0) pspec = 0.0;          /* angle > 90 => no specular    */
   else
    {spec =  nrml[0]*rflec[0] +           /* specular = dot product of    */
             nrml[1]*rflec[1] +           /* normal and reflected ray...*/
             nrml[2]*rflec[2];
     pspec = pow(spec,obj->pwr);          /* ... raised to power pwr.     */
     }

  intens = 15.0 * ( obj->ks*pspec +       /* calculate Phong intensity */
                    obj->kd*diff  +       /* (0..1) and convert to the */
                    obj->ka*ambient) +    /* range 0..15 for PGA use.  */
                    obj->rnd*rand()/32768.0;
    }
 else                                     /* shad=1 => point in shadow */
    intens = 15.0*obj->ka*ambient + obj->rnd*rand()/32768.0;

    if (intens < 0.0) intens = 0.0;
     else                                 /* chop out of range values  */
      if (intens > 14.9) intens = 14.9;

  if (DITHER == ON)                       /* dither flag set ? */

    {dther = intens - (int)intens;        /* take fractional part. */
     if  ( (int)(15.0*dther+0.5) >        /* Use dither do decide  */
               dither4[x%4][y%4] )        /* to round up or down.  */
          intens += 1.0;
    }
  else
     intens += 0.5;          /* no dither, round to nearest integer */

  *c = (char)(16*obj->clr + (int)intens);
                                          /* 0..255 color value */
}


void rgbshade(c,obj,ray,lght,nrml,x,y,shad)
unsigned char *c;
OBJECT *obj;
VECTOR ray,lght,nrml;
int x,y,shad;

/*
   ================================================================
  | This function calculates the shade of a pixel as three intensity |
  | values  in  the range 0..255,  one for each of the red green and |
  | blue intensities, by applying the shading equation three times - |
  | once for each intensity.  The reapective ratios  with  which  an |
  | object reflects each of the three primaries is obtained from the |
  | object structure itself.                                         |
   ================================================================
*/

{VECTOR rflec;                  /* direction of reflected ray.     */
int i;
```

```
double spec,pspec,              /* specular reflection values      */
       diff,                    /* diffuse reflection value        */
       intens,                  /* final calculated intensity      */
       dther,                   /* used in dither matrix operation */

if ('shad)
 {
  diff = nrml[0]*lght[0] +            /* diffuse intensity =      */
         nrml[1]*lght[1] +            /* vector dot product of    */
         nrml[2]*lght[2],             /* normal and light vectors */

  if (diff < 0 0) diff = 0 0,         /* negative => angle > 90   */

  rflec[0] = lght[0]-ray[0],          /* calculate direction of   */
  rflec[1] = lght[1]-ray[1],          /* reflected ray for use in */
  rflec[2] = lght[2]-ray[2],          /* specular calculation     */
  normalize(rflec),                   /* must be unit vector      */

  if (diff == 0 0) pspec = 0 0,       /* angle > 90 => no specular */
   else
    {spec =  nrml[0]*rflec[0] +       /* specular = dot product of   */
             nrml[1]*rflec[1] +       /* normal and reflected ray    */
             nrml[2]*rflec[2],
     pspec = pow(spec,obj->pwr),      /*     raised to power pwr     */
     }
 }
for (i=0, i<3, i++)
{
 if (shad)
   intens = 255 0 * (1 0 - obj->ka*ambient),
  else
   intens = 255 0 *
                (1 0 - ( obj->ka*ambient +      /* Phong intensity */
                         obj->ks*pspec +
                         obj->cmy[i]*diff  +
                         obj->cmy[i]*ambient)),

    if (intens < 0 0) intens = 0 0,
     else                             /* chop out of range values  */
      if (intens > 254 9) intens = 254 9,

  if (DITHER == ON)                   /* dither flag set ? */
   {
     dther = intens - (int)intens,    /* take fractional part  */
     if  ( (int)(15 0*dther+0 5) >    /* Use dither do decide  */
               dither4[x%4][y%4] )    /* to round up or down   */
          intens += 1 0,
    }
  else
    intens += 0 5,          /* no dither, round to nearest integer */
  *(c+i) = (char)(16*obj->clr + (int)intens),
  }
}
```

```
/* PGA  DEPENDENT  MODULE
   ================================================================
   | The  raytracer functions in this module are all PGA dependent in |
   | that they all call one or more functions from  the  set  of  PGA |
   | library  routines  which  were  written  prior  to the raytracer |
   | implementation and which provide access  to  various  line/curve |
   | drawing capabilities of the Professional Graphics adapter card   |
   |                                                                  |
   |    Function          PGA functions called                        |
   |                                                                  |
   |  rendervolume       poly    -->  draws polygon from of points    |
   |                     prmfil  -->  fills polygons in current color |
   |                     color   -->  sets current color              |
   |                                                                  |
   |  pgatrace           rline   -->  reads line of pixels from screen|
   |                     wline   -->  writes line of pixels to screen |
   |                                                                  |
   |  loadpgafile        wline                                        |
   |                    Gscreen  -->  sends byte stream directly to   |
   |                                  PGA memory mapped I/O buffer    |
   |                                                                  |
   |  savecolors         lutrd   -->  reads look up table value       |
   |                                                                  |
   |  readcolors         lut     -->  sets look up table value        |
   |                                                                  |
   |  generatergbup      lut                                          |
   |                                                                  |
   |  generatergbdown    lut                                          |
   |                                                                  |
   |  initpga            init    -->  initializes the PGA card and    |
   |                                  switches monitor to PGA mode    |
   |                     vwport  -->  defines screen viewport         |
   |                     window  -->  defines viewplane window        |
   |                     "flood" -->  floods viewport in given color  |
   |                                                                  |
   |  quitpga       endgraphic   -->  switches monitor from PGA back  |
   |                                  to normal mode                  |
   ==================================================================  */

#include   <stdio.h>
#include   <malloc.h>
#include   "pga.h"          /* pga library functions header file     */
#include   "global.h"       /* global parameters, flags & variables  */
#include   "function.h"     /* function prototype declarations       */

extern int fillcolor,


void rendervolume(v,c)
POINT v[8],
int c,                          /* bounding volume fill color   */
/* ================================================================
   | Takes specified bounding volume (transformed and projected)  and |
   | renders  it  on  the screen (using PGA library functions) as six |
   | filled polygons, filled in color c                               |
   ================================================================ */
{int dummy[3],
  prmfil(on),               /* ploygon fill on   */
  color(c),                 /* current color = c */

   /* render six polygon faces of  bounding volume on screen,
      filled with fillcolor  (volume for cone really only has
      five, 6th one consists of a single point - the apex       */

  poly(4,v[0][0],v[0][1] v[1][0] v[1][1],v[2][0],v[2][1],v[3][0],v[3][1]),
  poly(4,v[0][0],v[0][1],v[4][0],v[4][1],v[5][0],v[5][1],v[1][0],v[1][1])
  poly(4,v[0][0],v[0][1],v[3][0],v[3][1],v[7][0],v[7][1],v[4][0],v[4][1]),
  poly(4,v[6][0],v[6][1],v[5][0],v[5][1],v[1][0],v[1][1],v[2][0],v[2][1]),
```

```
  poly(4,v[6][0],v[6][1],v[7][0],v[7][1],v[4][0] v[4][1],v[5][0],v[5][1]),
  poly(4,v[6][0],v[6][1],v[2][0],v[2][1],v[3][0],v[3][1],v[7][0],v[7][1]),
  lutrd(25,dummy),
  prmfil(off),              /* polygon fill off */
}


void pgatrace(scene)
OBJECT *scene,
{unsigned char *bf,*fbuf,*ptr,id,
int x,y,cnt,

 bf = (unsigned char *)malloc(vx2-vx1+1),
 if (FILEOUT == ON)
  {if (COMPRESS == ON)
     fbuf = (unsigned char *)malloc(2*(vx2-vx1+1)),
   else
    {id = 0xff,     /* write id byte to indicate uncompressed format */
     fwrite((char *)&id,1,1,outfile),
     fwrite((char *)&vx1,sizeof(int),1,outfile),   /* write viewport */
     fwrite((char *)&vx2,sizeof(int),1,outfile),
     fwrite((char *)&vy1,sizeof(int),1,outfile),
     fwrite((char *)&vy2,sizeof(int),1,outfile),
     }
    }
 for (y=vy2, y>=vy1, y--)
  { ptr = bf,
    if ((PIXELBUFFER == ON) && (SCREEN == ON))
     {rline(y,vx1,vx2,bf),
      for (x=vx1, x<=vx2, x++,ptr++)
        if (*ptr == fillcolor) raycast(scene,x,y,ptr),
         else *ptr = background,
       }
     else
      for (x=vx1, x<=vx2, x++,ptr++)
        raycast(scene,x,y,ptr),

     if (SCREEN == ON)
       wline(y,vx1,vx2,bf),

     if (FILEOUT == ON)
      {if (COMPRESS == ON)
        {cnt = linecompress(bf,fbuf,y,vx1,vx2),
         fwrite(fbuf,1,cnt,outfile),
         }
        else
         fwrite(bf,1,vx2-vx1+1,outfile),
        }
   }
  free(bf),
  if (FILEOUT == ON)
   { fclose(outfile),
     if (COMPRESS == ON)
       free(fbuf),
     }
}

int readcolors(str)
char *str,
/* ==========================================================
  | This function loads the active look up table of the PGA with the |
  | 256 red, green, & blue stored as integers in the file str  The |
  | integer stores the values as  -                                |
  |                                                                |
  |       red   -->  bits 11 10 9 8                                |
  |       green -->  bits  7  6 5 4                                |
  |       blue  -->  bits  3  2 1 0                                |
   ========================================================== */
```

```
{FILE *fp;
int i,r,g,b,*bf;

 if ((PGA == OFF) || (SCREEN == OFF)) return 0;

 if ((fp = fopen(str,"rb")) == NULL) return -1;

 bf = (int *)malloc(256*sizeof(int));
 fread((char *)bf,sizeof(int),256,fp);
 fclose(fp);

 for (i=0; i<256; i++)
  {r = (bf[i] & 0x0f00) >> 8;
   g = (bf[i] & 0x00f0) >> 4;
   b =  bf[i] & 0x000f;
   lut(i,r,g,b);
   }
return 1;
}


int savecolors(str)
char *str;
/* ============================================================
   | This function performs the reverse process of readcolors ie. it  |
   | saves each of the 256 12-bit entries of the current PGA look up  |
   | table as 256 integers stored in the file str.                    |
    ============================================================ */
{FILE *fp;
int i,rgb[3],*bf;

 if ((PGA == OFF) || (SCREEN == OFF)) return 0;

 if ((fp = fopen(str,"wb")) == NULL) return -1;

 bf = (int *)malloc(256*sizeof(int));

 for (i=0; i<256; i++)
  {lutrd(i,rgb);
   bf[i] = (rgb[0] << 8) | (rgb[1] << 4) |  rgb[2];
   }
 fwrite((char *)bf,sizeof(int),256,fp);
 fclose(fp);
return 1;
}
```

```
void generatergbup(c,r,g,b)
int c,r,g,b;
/*
   ============================================================
  | Given a color group (0..15) and a red, green & blue color,  this |
  | function  generates  a  set  of  16  shades  of the given color, |
  | starting with black (rgb = 000).  The shades are  calculated  by |
  | following  a  line through an imaginary RGB cube,  consisting of |
  | 4096 subcubes (16x16x16) each of which corresponds to one of the |
  | 4096 PGA red,  green & blue combinations.  The line is  followed |
  | from  origin  (lower  left  corner) through the cell specified by |
  | r,g, and b.                                                      |
   ============================================================
*/
```

```
{int i,
double x,y,z,max,

 if (SCREEN == ON)
 {
  if ((r > 0) || (g > 0) || (b > 0))
  {max = r,
   if (g >= max)  max = g,
   if (b >= max)  max = b,
   x = r/max,
   y = g/max,
   z = b/max,

   for (i=0, i<16, i++)
    lut(16*c+i,(int)(i*x+0 5),(int)(i*y+0 5),(int)(i*z+0 5)),
   }
 }
}


void generatergbdown(c,r,g,b)
int c,r,g,b,
/*
   ==========================================================
  | Like generatergbup, this function generates 16 shades of a color |
  | by following a line through an imaginary RGB cube  In this case  |
  | however, the line is followed from the upper right corner of the |
  | cube (rgb = 111) down through the cell specified by r, g, and b  |
   ==========================================================
*/
{int i,
double x,y,z,max,
if (SCREEN == ON)
 {
 if ((r > 0) || (g > 0) || (b > 0))
  {max = r,
   if (g >= max)  max = g,
   if (b >= max)  max = b,
   x = r/max,
   y = g/max,
   z = b/max,
   for (i=0, i<16, i++)
    lut(16*c+i,(int)(15 5-i*x),(int)(15 5-i*y),(int)(15 5-i*z)),
   }
 }
}




void loadpgafile(str)
char * str,
/*
   ==========================================================
  | Displays an image on screen,  read from a file  in  PGA  format  |
  | First  determines  if  the file is in compressed or uncompressed |
  | format by looking at the first byte  -                           |
  |                                                                  |
  |     D9 ==> compressed                                            |
  |     FF ==> uncompressed                                          |
  |                                                                  |
  | see linecompress function in  RTRACE C  module  for  description |
  | of compressed & uncompressed formats                             |
   ==========================================================  */
```

```
{char *lne,
unsigned char id,
int i=25,fh,x1,x2,y1,y2,y,len,
long length,
FILE *fp,

if (((fp = fopen(str,"rb")) != NULL))
 {
  fread(&id,1,1,fp),                    /* first byte D9 or FF        */
  if (id == 0xD9)
   { lne = malloc(i),                   /* D9 ==> compressed          */
     fh = fileno(fp),
     length = filelength(fh) ,
     fseek(fp,(long)0,SEEK_SET),        /* put byte back - its data   */
     while (length > (long)i)
       {fread(lne,1,i,fp),              /* read file in blocks of i   */
        Gscreen(lne,i),                 /* and send directly to PGA   */
        length -= (long)i,
        }
      fread(lne,1,(int)length,fp),
      Gscreen(lne,(int)length),
      fclose(fp),
      free(lne),
     }
  else
   if (id == 0xFF)                      /* FF  ==> uncompressed       */
    {                                   /* NOTE  FF not part of data  */
     fread((char *)&x1,sizeof(int),1,fp),          /* read viewport */
     fread((char *)&x2,sizeof(int),1,fp),
     fread((char *)&y1,sizeof(int),1,fp),
     fread((char *)&y2,sizeof(int),1,fp),
     len = x2-x1+1,
     lne = malloc(len),

     for (y=y2, y>=y1, y--)
      {fread(lne,1,len,fp),               /* read line & display it */
       wline(y,x1,x2,lne),
       }
     free(lne),
     fclose(fp),
     }
   }
}


void initpga()
{
  if (SCREEN == ON)
   {
    init(),
    vwport(vx1,vx2,vy1,vy2),
    window(wx1,wx2,wy1,wy2),
    flood(fillcolor^0xffff),
   }
}


void quitpga()
{
 if (SCREEN == ON)
  endgraphic(),
}
```

Bibliography

AMAN84 Amantatides, J , *Ray Tracing With Cones*, COMPUTER GRAPHICS 1984 VOL 18 #3 JULY PP 129-135

ANTO81 Anton, H , *Elementary Linear Algebra*, JOHN WILEY & SONS INC 1981

APOD89 Apodaca, T , *The Renderman Interface*, BYTE 1989 VOL 14 #4 APRIL PP 267-276

APPE67 Appel, A , *The Notion Of Quantitative Invisibility And The Machine Rendering Of Solids*, PROC ACM NATIONAL CONFERENCE (OCT) NEW YORK 1967 ACM NEW YORK PP 387-393

APPE68 Appel, A , *Some Techniques For Shading Machine Renderings Of Solids*, THOMPSON BOOKS WASHINGTON D C 1968 PP 37-45

ARNA87 Arnaldi, B , Thierry P and Bouatouch K , *A New Space Subdivision Method For Ray Tracing CSG Modelled Scenes*, VISUAL COMPUTING GERMANY, SPRINGER VERLAG 1987 VOL 3 #2 AUGUST PP 98-108

ARVO87 Arvo, J and Kirk, D , *Fast Ray Tracing By Ray Classification*, COMPUTER GRAPHICS 1987 VOL 21 #4 JULY PP 55-64

BLIN76 Blinn, F J and Newell, M E , *Texture And Reflection In Computer Generated Images*, COMMUNICATIONS OF THE ACM 1976 VOL 18 #10 OCTOBER PP 542-547

BLIN77 Blinn, J F , *Models Of Light Reflection For Computer Synthesized Pictures*, COMPUTER GRAPHICS 1977 VOL 11 #2 PP 192-198

BLIN80 Blinn, J F , Carpenter, J and Whitted, T , *Scan Line Methods For Displaying Parametrically Defined Surfaces*, COMMUNICATIONS OF THE ACM 1980 VOL 23 #1 JANUARY PP 23-34

BOUK70 Bouknight, W J , *A Procedure For Generation Of Three-Dimensional Halftoned Computer Graphics Representations*, COMMUNICATIONS OF THE ACM 1970 VOL 13 #9 SEPTEMBER PP 527-536

BOUV85 Bouville, C , *Bounding Ellipsoids For Ray-Fractal Intersection*, COMPUTER GRAPHICS 1985 VOL 19 #3

BOYS82 Boyse J W and Gilchrist J E , *GMsolid Interactive Modelling For Design And Analysis Of Solids*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 VOL 2 #2 MARCH PP 86-97

BRON84 Bronsvoort, W F , Van Wijk, J J and Jansen, F W , *Two Methods For Improving The Efficiency Of Ray Casting In Solid Modelling*, COMPUTER AIDED DESIGN 1984 VOL 16 #1 JANUARY PP 51-55

BROW82 Brown C M , *PADL-2 A Technical Summary*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 VOL 2 #2 MARCH PP 69-84

BUIT75 Bui-Tuong, Phong , *Illumination For Computer Generated Images*, COMMUNICATIONS OF THE ACM 1975 VOL 18 PP 311-317

CARP82 Carpenter L , Fournier A and Fussel D , *Computer Rendering Of Stochastic Models*, COMMUNICATIONS OF THE ACM 1982 VOL 25 #7 JUNE PP 371-384

CATM74 Catmull E , *A Subdivision Algorithm For Computer Display Of Curved Surfaces*, UNIV UTAH COMPUTER SCIENCE DEPT 1974 DECEMBER UTEC-CSC-74-133

CATM80 Catmull, E , *Computer Display Of Curved Surfaces*, TUTORIAL AND SELECTED READINGS IN INTERACTIVE COMPUTER GRAPHICS 1980 IEEE PP 309-315 (H FREEMAN ED)

CLEA83 Cleary, J G , Wyvill, G M , Vatti, R and Birtwistle G M , *Multiprocessor Ray Tracing*, TECH REPORT DEPT COMPUTER SCIENCE CALGARY UNIVERSITY 1983 REPORT # 83/128/17 OCTOBER

COHE85 Cohen, M F and Greenberg, D P , *The Hemi-Cube A Radiosity Solution For Complex Enviornments*, COMPUTER GRAPHICS 1985 VOL 19 #3 PP 31-41

COOK81 Cook, R L and Torrance, K , *A Reflectance Model For Computer Graphics*, COMPUTER GRAPHICS SIGGRAPH '81 1981 VOL 15 #3 AUGUST PP 307-316

COOK88 Cook, R L , *A Reflection Model For Realistic Image Synthesis*, MASTERS THESIS CORNELL UNIVERSITY ITHACA NY DECEMBER '88

CROW77a Crow, F C , *Shadow Algorithms For Computer Graphics*, COMPUTER GRAPHICS 1977 VOL 11 #3 JULY PP 242-248

CROW77b Crow, F C , *The Aliasing Problem In Computer Shaded Images*, COMMUNICATIONS OF THE ACM 1977 VOL 20 #11 PP 40-48

CROW81 Crow, F C , *A Comparison Of Antialiasing Techniques*, IEEE COMPUTER GRAPHICS & APPLICATIONS 1981 VOL 1 #1 JANUARY

DADO82 Dadoun, N , Kirkpatrick, D and Walsh J , *Hierarchical Approaches To Hidden Surface Intersection Testing*, PROCEEDINGS OF GRAPHICS INTERFACE '82 1982 MAY PP 49-56

DEGU86 Deguchi, H , Shirakawa, I and Omura, K , *A Tree-Structured Parallel Processing System For Image Generation By Ray Tracing*, SYSTEMS AND COMPUTING (USA) 1986 VOL 17 #12 DECEMBER PP 51-62

DIPP84 Dippe, M and Swensen, J , *An Adaptive Subdivision Algorithm And Parallel Architecture For Realistic Image Synthesis*, COMPUTER GRAPHICS 1984 VOL 18 #3 JULY PP 149-158

EDWA82 Edwards, B E , *Implementation Of A Ray Tracing Algorithm For Rendering Superquadric Solids*, MASTERS THESIS RENSSELAER POLYTECHNIC INST , TROY, NEW YORK 1982 DECEMBER

ERDE89 Erdelsky, Philip, J , *An Efficient Sorting Algorithm For Sorting Linked Lists*, THE C USERS JOURNAL 1989 VOL 7 # 4 MAY PP 89-91

FOLE84 Foley, J D and Van Dam, A , *Fundamentals Of Interactive Computer Graphics*, ADDISON-WELSEY SYSTEMS PROGRAMMING SERIES 1984 ISBN 0-201-14468-9

FUJI86 Fujimoto, A , *ARTS Accelerated Ray Tracing System*, IEEE COMPUTER GRAPHICS & APPLICATIONS 1986 APRIL PP 16-26

GERV86 Gervautz, M , *Three Improvements Of The Ray Tracing Algorithm For CSG Trees*, COMPUTING & GRAPHICS (GB) 1986 VOL 10 #4 PP 333-339

GOLD86 Goldsmith, J and Salmon, J , *Automatic Creation Of Object Hierarchies*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1986 VOL 7 #5 PP 14-20

GREE78 Greenberg, D P , Atherton, P R and Weiler, K J , *Polygon Shadow Generation*, COMPUTER GRAPHICS 1978 VOL 12 #3 PP 275-281

GREE79 Greenberg, D P and Kay, D S , *Transparency For Computer Synthesized Pictures*, COMPUTER GRAPHICS SIGGRAPH '79 1979 VOL 13 #2 AUGUST PP 158-164

HANR83 Hanrahan, P , *Ray Tracing Algebraic Surfaces*, COMPUTER GRAPHICS 1983 VOL 17 #3 JULY PP 83-90

HECK84 Heckbert, P S and Hanrahan, P , *Beam Tracing Polygonal Objects*, COMPUTER GRAPHICS 1984 VOL 18 #3 PP 119-127 JULY

HENN89 Henning, E , *Intel's Risc Revolution Of The '90s*, PC USER 1989 #103 29 MARCH PP 40-52

HIGD74 Higdon, C E , *An Optical Ray Tracing Program*, NAVAL ORDNANCE LAB WHITE OAK MD USA 1974 REPORT # NOLTR-74-70

JANS85 Jansen, F W , *Data Structures For Ray Tracing*, COMPUTER GENERATED IMAGES (PROC GRAPHICS INTERFACE '85) 1985 MAY 27-31 PP 57-73

JOY86 Joy, K I and Bhetanabhotla, M N , *Ray Tracing Parametric Surface Patches Utilizing Numerical Techniques And Ray Coherence*, COMPUTER GRAPHICS 1986 SIGGRAPH '86 AUGUST PP 18-22

KAJI83a Kajiya, J T , *New Techniques For Ray Tracing Procedurally Defined Objects*, COMPUTER GRAPHICS 1983 VOL 17 #3 JULY PP 91-102

KAJI83b Kajiya, J T , *Ray Tracing Parametric Patches*, COMPUTER GRAPHICS 1983 VOL 17 #3 PP 91-102

KAJI84 Kajiya, J T and Von Hersen, B P , *Ray Tracing Volume Densities*, COMPUTER GRAPHICS 1984 VOL 18 #3 JULY PP 165-174

KAY84 Kay, T L and Kajiya, J T , *Ray Tracing Complex Scenes*, COMPUTER GRAPHICS 1984 VOL 20 #4 PP 269-278

LANC78 Lance, W , *Casting Curved Shadows On Curved Surfaces*, COMPUTER GRAPHICS 1978 VOL 12 #3 AUGUST PP 270-274

LEE85 Lee, M , Redned, R A and Uselton, S,P , *Statistically Optimized Sampling For Distributed Ray Tracing*, COMPUTER GRAPHICS 1985 VOL 19 #3 JULY PP 61-67

LEVI76 Levin, J , *A Parametric Algorithm For Drawing Pictures Of Solid Objects Composed Of Quadric Surfaces*, COMMUNICATIONS OF THE ACM 1976 VOL 19 #10 OCTOBER PP 555-563

MAHL72 Mahl, R , *Visible Surface Algorithm For Quadric Patches*, IEEE TRANSACTIONS ON COMPUTERS 1972 C-21 JANUARY PP 1-4

MAND82 Mandelbrot B B , *The Fractal Geometry Of Nature*, FREEMAN SAN FRANCISCO 1982

MAX86 Max, N L , *Atmospheric Illumination And Shadows*, COMPUTER GRAPHICS SIGGRAPH '86 DALLAS 1986 VOL 20 #4 AUGUST PP 117-124

MAXW46 Maxwell, E A , *Methods Of Plane Projective Geometry Based On The Use Of General Homogenoug Coordinates*, CAMBRIDGE UNIVERSITY PRESS 1951 CAMBRIDGE

MAXW86 Maxwell, G M , *Calculations Of The Radiation Configuration Using Ray Casting*, COMPUTER AIDED DESIGN (GB) 1986 VOL 18 #7 SEPTEMBER PP 371-379

MITC87 Mitchell, D P , *Generating Antialiased Images At Low Sampling Densities*, COMPUTER GRAPHICS 1987 VOL 21 #4 JULY PP 65-71

MYER82 Myers W , *An Industrial Perspective On Solid Modelling*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 VOL 2 #2 MARCH PP 86-97

NAGE71 Nagel, R and Goldstein, R A , *3-D Visual Simulation*, SIMULATION 1971 JANUARY PP 25-31

NEMO86 Nemoto, K and Omachi, T , *An Adaptive Subdivision By Sliding Boundary Surfaces For Fast Ray Tracing*, PROC OF GRAPHICS INTERFACE '86 & VISION INTERFACE '86 1986 MAY WEIN, M AND KIDD, E M (EDS) -

NEWE72 Newell, M E , Newell, R G and Sancha, T L , *A New Approach To The Shaded Picture Problem*, PROCEEDINGS ACM NATIONAL CONFERENCE 1972 PP 443

NEWE78 Blinn, J F and Newell, M E , *Clipping Using Homogenous Coordinates*, COMPUTER GRAPHICS SIGGRAPH '78 1978 VOL 12 #3 AUGUST PP 245-251

NEWM79 Newman W,M and Sproul R F , *Principles Of Interactive Computer Graphics*, MC GRAW HILL 2ND EDITION 1979

NISH86 Nishita, T and Nakamae, E , *Continous Tone Representation Of Three-Dimensional Objects Illuminated By Sky Light*, COMPUTER GRAPHICS SIGGRAPH '86 DALLAS 1986 VOL 20 #4 AUGUST PP 125-132

PEAC86 Peachey, D R , *PORTRAY-An Image Synthesis System*, PROC GRAPHICS INTERFACE '86 & VISION INTERFACE, VANCOUVER 1986 26-30 MAY PP 37-42 WEIN, M AND KIDD, E (EDS)

PLUN85 Plunkett, D J and Bailey, M J , *The Vectorization Of A Ray Tracing Algorithm For Improved Execution Speed*, IEEE COMPUTER GRAPHICS & APPLICATIONS 1985 VOL 5 #8 AUGUST PP 52-60

PORT84 Porter, T , Cook, R L and Carpenter, L , *Distributed Ray Tracing*, COMPUTER GRAPHICS 1984 VOL 18 #3 JULY PP 137-145

PULL87 Pulleyblank, R W , *The Feasibility Of A VLSI Chip For Ray Tracing Bicubic Patches*, IEEE COMPUTER GRAPHICS & APPLICATIONS 1987 VOL 7 #3 MARCH PP 33-44

REQU80 Requicha A , *Representations For Rigid Solids Theory, Methods And Systems*, COMPUTING SURVEYS 1980 VOL 12 #4 DECEMBER PP 437-464

REQU82 Requicha A and Voelcker H B , *Solid Modelling A Historical Summary And Contemporary Assessment*, IEEE COMPUTER GRAPHICS AND APPLICATIONS 1982 VOL 2 #2 MARCH PP 9-24

ROBE72 Roberts, B C jr and Bebbs, E H , *Atmospheric Ray Tracing*, PROGRAM OF 84TH MEETING OF ACOUSTICAL SOCIETY OF AMERICA 1972 P 68 NOVEMBER '72 MIAMI LINDSAY, R (ED)

ROTH82 Roth, S D , *Ray Casting For Modelling Solids*, COMPUTER GRAPHICS AND IMAGE PROCESSING 1982 VOL 18 PP 109-144

RUBI80 Rubin, S M and Whitted, T , *A 3-Dimensional Representation For Fast Rendering Of Complex Scenes*, COMPUTER GRAPHICS 1980 VOL 14 #3 JULY PP 110-116

RUSH86 Rushmeier, H E , *Extending The Radiosity Method To Transmitting And Specularly Reflecting Surfaces*, MASTERS THESIS CORNELL UNIVERSITY ITHACA N Y 1986

SEDE84 Sederberg, T W and Anderson, D C , *Ray Tracing Of Steiner Patches*, COMPUTER GRAPHICS 1984 VOL 18 #3 PP 159-164

SHAO88 Shao Min-Zhi, Peng Qun-Sheng and Liang You-Dong, *A New Radiosity Approach By Procedural Refinements For Realistic Image Synthesis*, COMPUTER GRAPHICS SIGGRAPH '88 1988 VOL 22 #4 PP 93-102

SHIN87 Shinya, M , Takahashi, T and Naito, S , *Principles And Applications Of Pencil Tracing*, COMPUTER GRAPHICS 1987 VOL 21 #4 JULY PP 45-54

SPEE85 Speer, L , Derose T D and Barsky, B A , *A Theoretical And Empirical Analysis Of Coherent Ray Tracing*, COMPUTER GENERATED IMAGES (PROC GRAPHICS INTERFACE '85) 1985 MAY 27-31 PP 11-25

SUTH74 Sutherland, I E , Sproull, R F and Schumacker, R A , *A Characterization Of Ten Hidden Surface Removal Algorithms*, COMPUTING SURVEYS 1974 VOL 6 #1 MARCH PP 1-55

TORR66 Torrance, K E and Sparrow, E M , *Polarization, Direction, Distribution & Off-Specular Peak Phenomena In Light Reflected From Roughened Surfaces*, JOURNAL OF THE OPTICAL SOCIETY OF AMERICA 1966 VOL 57 #7 JULY PP 916-925

TOTH85 Toth, D L , *On Ray Tracing Parametric Surfaces*, COMPUTER GRAPHICS 1985 VOL 19 #3 JULY PP 171-179

VANW84 Van Wijk, J J , *Ray Tracing Objects Defined By Sweeping A Sphere*, PROC EUROGRAPHICS '84 1984 (BO, K AND TUCKER, H A (EDS) PP 73-82

VERR85 Verroust, A , *Visualisation Method For Constructive Solid Geometry Using Polygon Clipping*,

VOEL77 Voelcker H B and Requicha A G , *Geometric Modelling Of Mechanical Parts And Processes*, COMPUTER 1977 VOL 10 #12 DECEMBER PP 48-57

WALL87 Wallace, J R , Greenberg, D P and Cohen, M F , *A Two Pass Solution To The Rendering Equation A Synthesis Of Ray Tracing And Radiosity Methods*, COMPUTER GRAPHICS 1987 VOL 21 #4 JULY PP 311-320

WARD88 Ward G J , Rubinstein F M and Clear R D , *A Ray Tracing Solution For Diffuse Interreflection*, COMPUTER GRAPHICS SIGGRAPH '88 1988 VOL 22 #4 PP 85-92

WATK70 Watkins, G S , *A Real Time Visible Surface Algorithm*, UNI UTAH COMP SCIENCE DEPT NTIS AD-762 004 1970 JUNE UTEC-CSC-70-101

WEGH84 Weghorst, H , Hooper, G and Greenberg, D P , *Improved Computational Methods For Ray Tracing*, ACM TRANSACTIONS ON GRAPHICS 1984 VOL 3 PP 52-69

WEIS66 Weiss, R A , *A Package Of Programs To Draw Orthographic Views Of Combinations Of Planes & Quadric Surfaces*, JOURNAL OF THE ACM 1966 VOL 13 #2 APRIL PP 194

WHIT80 Whitted, T , *An Improved Illumination Model For Shaded Display*, COMMUNICATIONS OF THE ACM 1980 VOL 23 #6 JUNE PP 343-349

YOUS86 Youssef, Saul, *A New Algorithm For Object Orientated Ray Tracing*, COMPUTER VISION, GRAPHICS, AND IMAGE PROCESSING 1986 VOL 34 PP 125-137